# Oracle® Tuxedo CORBA Programming Reference





Oracle Tuxedo CORBA Programming Reference, Release 22c

G11331-01

Copyright © 1996, 2025, Oracle and/or its affiliates.

Primary Author: Preeti Gandhe
Contributing Authors: Tulika Das

Contributors: Maggie Li

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

| 1.1 OM    | GIDL Compiler Extensions  | 1-2  |
|-----------|---|------|
|           | IDL Compiler Constraints  | 1-2  |
| Implem    | entation Configuration File (ICF)                                   |      |
| 2.1 ICF   | Syntax  | 2-1  |
| 2.2 Sam   | ple ICF File  | 2-2  |
| 2.3 Crea  | ting the ICF File   | 2-3  |
| TP Fran   | nwork   |      |
| 3.1 A Si  | mple Programming Model  | 3-2  |
| 3.1.1     | Control Flow  | 3-3  |
| 3.1.2     | Object State Management   | 3-3  |
| 3.1.3     | Transaction Integration   | 3-3  |
| 3.1.4     | Object Housekeeping   | 3-3  |
| 3.1.5     | High-level Services   | 3-4  |
| 3.2 State | e Management  | 3-4  |
| 3.2.1     | Activation Policy   | 3-4  |
| 3.2.2     | Application-controlled Activation and Deactivation                  | 3-6  |
| 3.2       | 2.2.1 Explicit Activation   | 3-6  |
| 3.2       | 2.2.2 Self Deactivation   | 3-7  |
| 3.2.3     | Servant Lifetime  | 3-8  |
| 3.2       | 2.3.1 The Normal Case   | 3-8  |
| 3.2       | 2.3.2 Special Cases   | 3-9  |
| 3.2.4     | Saving and Restoring Object State                                   | 3-10 |
| 3.2       | 2.4.1 Note On Use of Constructors and Destructors for CORBA Objects | 3-10 |
| 3.3 Tran  | sactions  | 3-10 |
| 3.3.1     | Transaction Policies  | 3-10 |
| 3.3.2     | Transaction Initiation  | 3-11 |
| 3.3.3     | Transaction Termination   | 3-12 |
| 3.3.4     | Transaction Suspend and Resume                                      | 3-12 |
| 3.3.5     | Restrictions on Transactions  | 3-13 |



|     | 3.3.6 | SQL     | and Global Transactions                            | 3-14 |
|-----|-------|---------|--|------|
|     | 3.3.7 | Votin   | g on Transaction Outcome                           | 3-14 |
|     | 3.3.8 | Trans   | saction Timeouts                                   | 3-15 |
| 3.4 | IIOP  | Client  | Failover   | 3-15 |
|     | 3.4.1 | Settir  | ng The Retry Policy                                | 3-15 |
|     | 3.4   | .1.1    | MIB Support  | 3-16 |
|     | 3.4.2 | Initiat | ing IIOP Client Failover                           | 3-16 |
|     | 3.4   | .2.1    | IIOP Client Failover Limitations                   | 3-16 |
| 3.5 | Webl  | _ogic ( | CORBA Clustering and Load Balancing Support        | 3-16 |
| 3.6 | Paral | lel Ob  | jects  | 3-17 |
| 3.7 | TP F  | ramev   | vork API   | 3-18 |
|     | 3.7.1 | Serve   | er Interface                                       | 3-19 |
|     | 3.7   | .1.1    | C++ Declarations                                   | 3-19 |
|     | 3.7   | .1.2    | Server::create_servant()                           | 3-21 |
|     | 3.7   | .1.3    | Server::initialize()                               | 3-24 |
|     | 3.7   | .1.4    | Server::release()                                  | 3-27 |
|     | 3.7.2 | Tobj_   | ServantBase Interface                              | 3-30 |
|     | 3.7   | .2.1    | C++ Declaration (in Tobj_ServantBase.h)            | 3-31 |
|     | 3.7   | .2.2    | Tobj_ServantBase:: activate_object()               | 3-31 |
|     | 3.7   | .2.3    | Tobj_ServantBase::_add_ref()                       | 3-34 |
|     | 3.7   | .2.4    | Tobj_ServantBase::deactivate_object()              | 3-35 |
|     | 3.7   | .2.5    | Tobj_ServantBase::_is_reentrant()                  | 3-39 |
|     | 3.7   | .2.6    | Tobj_ServantBase::_remove_ref()                    | 3-40 |
|     | 3.7.3 | TP In   | terface  | 3-41 |
|     | 3.7   | .3.1    | Usage Notes  | 3-41 |
|     | 3.7   | .3.2    | TP::application_responsibility()                   | 3-42 |
|     | 3.7   | .3.3    | TP::bootstrap()                                    | 3-43 |
|     | 3.7   | .3.4    | TP::close_xa_rm()                                  | 3-44 |
|     | 3.7   | .3.5    | TP::create_active_object_reference()               | 3-46 |
|     | 3.7   | .3.6    | TP::create_object_reference()                      | 3-48 |
|     | 3.7   | .3.7    | TP::deactivateEnable()                             | 3-50 |
|     | 3.7   | .3.8    | TP::get_object_id ()                               | 3-52 |
|     | 3.7   | .3.9    | TP::get_object_reference()                         | 3-53 |
|     | 3.7   | .3.10   | TP::open_xa_rm()                                   | 3-54 |
|     | 3.7   | .3.11   | TP::orb()  | 3-56 |
|     | 3.7   | .3.12   | TP::register_factory()                             | 3-57 |
|     | 3.7   | .3.13   | TP::unregister_factory()                           | 3-58 |
|     | 3.7   | .3.14   | TP::userlog()                                      | 3-60 |
| 3.8 | CosT  | ransa   | ctions::TransactionalObject Interface Not Enforced | 3-61 |
| 3.9 | Error | Cond    | itions, Exceptions, and Error Messages             | 3-61 |
|     | 3.9.1 | Exce    | ptions Raised by the TP Framework                  | 3-61 |
|     | 3.9.2 | Exce    | ptions in the Server Application Code              | 3-61 |



| 3.9.3     | 9.2.1 Example Exceptions and Transactions                   | 3-6<br>3-6 |
|-----------|---|------------|
|           | ·   | 3-6        |
| 3.9.4     | Restriction of Nested Calls on CORBA Objects                | 3-0        |
| CORBA     | A Bootstrapping Programming Reference                       |            |
| 4.1 Why   | Bootstrapping Is Needed                                     | 4-         |
| 4.2 Տսրբ  | ported Bootstrapping Mechanisms                             | 4-         |
| 4.3 Orac  | cle Bootstrapping Mechanism                                 | 4-         |
| 4.3.1     | How Bootstrap Objects Work                                  | 4-         |
| 4.3.2     | Types of Oracle Remote Clients Supported                    | 4-         |
| 4.3.3     | Capabilities and Limitations                                | 4-         |
| 4.4 Boot  | tstrap Object API   | 4-         |
| 4.4.1     | Tobj Module   | 4-         |
| 4.4.2     | C++ Mapping   | 4-         |
| 4.4.3     | Java Mapping  | 4-         |
| 4.4.4     | Automation Mapping  | 4-         |
| 4.4.5     | C++ Member Functions  | 4-         |
| 4.4       | 4.5.1 Tobj_Bootstrap  | 4-         |
| 4.4       | 4.5.2 Tobj_Bootstrap::register_callback_port                | 4-1        |
| 4.4       | 4.5.3 Tobj_Bootstrap::resolve_initial_references            | 4-1        |
| 4.4       | 4.5.4 Tobj_Bootstrap::destroy_current()                     | 4-1        |
| 4.4.6     | Java Methods  | 4-1        |
| 4.4.7     | Automation Methods  | 4-1        |
| 4.4       | 4.7.1 Initialize  | 4-1        |
| 4.4       | 4.7.2 CreateObject  | 4-1        |
| 4.4       | 4.7.3 DestroyCurrent  | 4-2        |
| 4.5 Boot  | tstrap Object Programming Examples                          | 4-2        |
| 4.5.1     | Visual Basic Client Example: Using the Bootstrap Object     | 4-2        |
| 4.6 Inter | operable Naming Service Bootstrapping Mechanism             | 4-2        |
| 4.6.1     | Introduction  | 4-2        |
| 4.6.2     | INS Object References                                       | 4-2        |
| 4.6.3     | INS Command-line Options                                    | 4-2        |
| 4.6.4     | INS Initialization Operations                               | 4-2        |
| 4.6.5     | INS Object URL Schemes                                      | 4-2        |
| 4.6       | 5.5.1 IOR URL Scheme  | 4-2        |
| 4.6       | 6.5.2 corbaloc URL Scheme                                   | 4-2        |
|           | 6.5.3 corbaname URL Scheme                                  | 4-2        |
| 4.6.6     | Getting a FactoryFinder Object Reference Using INS          | 4-2        |
| 4.6.7     | Getting a PrincipalAuthenticator Object Reference Using INS | 4-2        |
| 4.6.8     | Getting a TransactionFactory Object Reference Using INS     | 4-3        |



#### 5 FactoryFinder Interface

| 5.1 | Capabilitie | es, Limitations, and Requirements               | 5-1  |
|-----|-------------|---|------|
| 5.2 | Functiona   | ll Description                                  | 5-2  |
| į   | 5.2.1 Loc   | ating a FactoryFinder                           | 5-2  |
| į   | 5.2.2 Reg   | gistering a Factory                             | 5-3  |
|     | 5.2.2.1     | C++ Mapping                                     | 5-4  |
| į   | 5.2.3 Loc   | ating a Factory                                 | 5-4  |
|     | 5.2.3.1     | CORBAservices Naming Service Module OMG IDL     | 5-5  |
|     | 5.2.3.2     | CORBAservices Life Cycle Service Module OMG IDL | 5-5  |
|     | 5.2.3.3     | Tobj Module OMG IDL                             | 5-6  |
|     | 5.2.3.4     | Locating Factories in Another Domain            | 5-7  |
|     | 5.2.3.5     | Why Use Oracle Tuxedo CORBA Extensions?         | 5-7  |
| į   | 5.2.4 Crea  | ating Application Factory Keys                  | 5-8  |
|     | 5.2.4.1     | Names Library Interface Pseudo OMG IDL          | 5-8  |
|     | 5.2.4.2     | C++ Mapping                                     | 5-13 |
|     | 5.2.4.3     | Java Mapping                                    | 5-14 |
| 5.3 | C++ Mem     | ber Functions and Java Methods                  | 5-14 |
| į   | 5.3.1 Cos   | sLifeCycle::FactoryFinder::find_factories       | 5-15 |
|     | 5.3.1.1     | Synopsis  | 5-15 |
|     | 5.3.1.2     | C++ Mapping                                     | 5-15 |
|     | 5.3.1.3     | Java Mapping                                    | 5-15 |
|     | 5.3.1.4     | Parameter                                       | 5-16 |
|     | 5.3.1.5     | Exception                                       | 5-16 |
|     | 5.3.1.6     | Description                                     | 5-16 |
|     | 5.3.1.7     | Return Values                                   | 5-16 |
| į   | 5.3.2 Tobj  | j::FactoryFinder::find_one_factory              | 5-16 |
|     | 5.3.2.1     | Synopsis  | 5-17 |
|     | 5.3.2.2     | C++ Mapping                                     | 5-17 |
|     | 5.3.2.3     | Java Mapping                                    | 5-17 |
|     | 5.3.2.4     | Parameter                                       | 5-17 |
|     | 5.3.2.5     | Exceptions                                      | 5-17 |
|     | 5.3.2.6     | Description                                     | 5-18 |
|     | 5.3.2.7     | Return Values                                   | 5-18 |
| į   | 5.3.3 Tobj  | j::FactoryFinder::find_one_factory_by_id        | 5-18 |
|     | 5.3.3.1     | Synopsis  | 5-18 |
|     | 5.3.3.2     | C++ Mapping                                     | 5-19 |
|     | 5.3.3.3     | Java Mapping                                    | 5-19 |
|     | 5.3.3.4     | Parameter                                       | 5-19 |
|     | 5.3.3.5     | Exceptions                                      | 5-19 |



|     | 5.3.3.6   | Description                               | 5-19 |
|-----|-----------|---|------|
|     | 5.3.3.7   | Return Values                             | 5-20 |
|     | 5.3.4 Tob | pj::FactoryFinder::find_factories_by_id   | 5-20 |
|     | 5.3.4.1   | Synopsis                                  | 5-20 |
|     | 5.3.4.2   | C++ Mapping                               | 5-20 |
|     | 5.3.4.3   | Java Mapping                              | 5-20 |
|     | 5.3.4.4   | Parameter                                 | 5-21 |
|     | 5.3.4.5   | Exceptions                                | 5-21 |
|     | 5.3.4.6   | Description                               | 5-21 |
|     | 5.3.4.7   | Return Values                             | 5-22 |
|     | 5.3.5 Tol | pj::Factoryfinder::list_factories         | 5-22 |
|     | 5.3.5.1   | Synopsis                                  | 5-22 |
|     | 5.3.5.2   | C++ Mapping                               | 5-22 |
|     | 5.3.5.3   | Java Mapping                              | 5-22 |
|     | 5.3.5.4   | Exception                                 | 5-22 |
|     | 5.3.5.5   | Description                               | 5-23 |
|     | 5.3.5.6   | Return Values                             | 5-23 |
| 5.4 | Automati  | on Methods                                | 5-23 |
|     | 5.4.1 DIT | Fobj_FactoryFinder.find_one_factory       | 5-23 |
|     | 5.4.1.1   | Synopsis                                  | 5-24 |
|     | 5.4.1.2   | MIDL Mapping                              | 5-24 |
|     | 5.4.1.3   | Automation Mapping                        | 5-24 |
|     | 5.4.1.4   | Parameters                                | 5-24 |
|     | 5.4.1.5   | Exceptions                                | 5-24 |
|     | 5.4.1.6   | Description                               | 5-24 |
|     | 5.4.1.7   | Return Values                             | 5-25 |
|     | 5.4.2 DIT | Fobj_FactoryFinder.find_one_factory_by_id | 5-25 |
|     | 5.4.2.1   | Synopsis                                  | 5-25 |
|     | 5.4.2.2   | MIDL Mapping                              | 5-25 |
|     | 5.4.2.3   | Automation Mapping                        | 5-25 |
|     | 5.4.2.4   | Parameters                                | 5-25 |
|     | 5.4.2.5   | Exceptions                                | 5-26 |
|     | 5.4.2.6   | Description                               | 5-26 |
|     | 5.4.2.7   | Return Values                             | 5-26 |
|     | 5.4.3 DIT | Fobj_FactoryFinder.find_factories_by_id   | 5-26 |
|     | 5.4.3.1   | Synopsis                                  | 5-26 |
|     | 5.4.3.2   | MIDL Mapping                              | 5-27 |
|     | 5.4.3.3   | Automation Mapping                        | 5-27 |
|     | 5.4.3.4   | Parameters                                | 5-27 |
|     | 5.4.3.5   | Exceptions                                | 5-27 |
|     | 5.4.3.6   | Description                               | 5-27 |
|     | 5.4.3.7   | Return Values                             | 5-28 |
|     |           |   |      |



| 5.4.4 DIT    | obj_FactoryFinder.find_factories  | 5-28 |
|--------------|---|------|
| 5.4.4.1      | Synopsis  | 5-28 |
| 5.4.4.2      | MIDL Mapping  | 5-28 |
| 5.4.4.3      | Automation Mapping  | 5-28 |
| 5.4.4.4      | Parameters  | 5-28 |
| 5.4.4.5      | Exception   | 5-28 |
| 5.4.4.6      | Description   | 5-28 |
| 5.4.4.7      | Return Values   | 5-29 |
| 5.4.5 DIT    | obj_FactoryFinder.list_factories  | 5-29 |
| 5.4.5.1      | Synopsis  | 5-29 |
| 5.4.5.2      | MIDL Mapping  | 5-29 |
| 5.4.5.3      | Automation Mapping  | 5-29 |
| 5.4.5.4      | Parameter   | 5-29 |
| 5.4.5.5      | Exception   | 5-29 |
| 5.4.5.6      | Description   | 5-30 |
| 5.4.5.7      | Return Values   | 5-30 |
| 5.5 Programm | ning Examples   | 5-30 |
| 5.5.1 Usir   | ng the FactoryFinder Object   | 5-30 |
| 5.5.1.1      | Registering a Reference to a Factory Object   | 5-31 |
| 5.5.1.2      | Obtaining a Reference to a FactoryFinder Object Using the CosLifeCycle::FactoryFinder Interface | 5-31 |
| 5.5.1.3      | Obtaining a Reference to a FactoryFinder Object Using the Extensions<br>Bootstrap object        | 5-31 |
| 5.5.2 Usir   | ng Extensions to the FactoryFinder Object   | 5-32 |
| 5.5.2.1      | Obtaining One Factory Using Tobj::FactoryFinder   | 5-32 |
| 5.5.2.2      | Obtaining One or More Factories Using Tobj::FactoryFinder                                       | 5-32 |
| Security Se  | rvice   |      |
| Transaction  | is Service  |      |
| Notification | Service   |      |
| Request-Le   | evel Interceptors   |      |
| CORBA Into   | erface Repository Interfaces  |      |
|              | e and Usage   | 10-2 |
|              | ogramming Information   | 10-2 |
|              |   |      |



|    | 10.2 | Buildi | ing Client Applications  | 10-4  |
|----|------|--------|--|-------|
|    | 10.3 | Gettir | ng Initial References to the InterfaceRepository Object                | 10-4  |
|    | 10.4 | Interf | ace Repository Interfaces  | 10-4  |
|    | 10   | 0.4.1  | Supporting Type Definitions  | 10-5  |
|    | 10   | 0.4.2  | IRObject Interface   | 10-6  |
|    | 10   | 0.4.3  | Contained Interface  | 10-6  |
|    | 10   | 0.4.4  | Container Interface  | 10-7  |
|    | 10   | ).4.5  | IDLType Interface  | 10-9  |
|    | 10   | 0.4.6  | Repository Interface   | 10-9  |
|    | 10   | 0.4.7  | ModuleDef Interface  | 10-10 |
|    | 10   | 0.4.8  | ConstantDef Interface  | 10-10 |
|    | 10   | 0.4.9  | TypedefDef Interface   | 10-11 |
|    | 10   | 0.4.10 | StructDef  | 10-11 |
|    | 10   | 0.4.11 | UnionDef   | 10-11 |
|    | 10   | 0.4.12 | EnumDef  | 10-12 |
|    | 10   | 0.4.13 | AliasDef   | 10-12 |
|    | 10   | 0.4.14 | PrimitiveDef   | 10-13 |
|    | 10   | 0.4.15 | StringDef  | 10-13 |
|    | 10   | 0.4.16 | WstringDef   | 10-14 |
|    | 10   | 0.4.17 | ExceptionDef   | 10-14 |
|    | 10   | 0.4.18 | AttributeDef   | 10-14 |
|    | 10   | 0.4.19 | OperationDef   | 10-15 |
|    | 10   | 0.4.20 | InterfaceDef   | 10-16 |
| 11 | Join | t Clie | ent/Servers  |       |
|    | 11.1 | Introd | luction  | 11-1  |
|    | 11   | 1.1    | Main Program and Server Initialization                                 | 11-2  |
|    | 11   | 1.2    | Servants   | 11-2  |
|    | 11   | 1.3    | Servant Inheritance from Skeletons                                     | 11-3  |
|    |      | 11.1.  | .3.1 C++ Example of Inheritance from Skeletons                         | 11-3  |
|    | 11   | 1.4    | Callback Object Models Supported                                       | 11-3  |
|    | 11   | 1.5    | Configuring Servers to Call Remote Joint Client/Server Objects         | 11-4  |
|    | 11   | 1.6    | Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only) | 11-5  |
|    | 11   | 1.7    | Preparing Callback Objects Using OracleWrapper Callbacks               | 11-6  |
|    |      | 11.1.  | .7.1 Using OracleWrapper Callbacks With C++                            | 11-6  |
|    | 11.2 | C++ (  | OracleWrapper Callbacks Interface API                                  | 11-8  |
|    | 11   | 2.1    | Callbacks  | 11-8  |
|    |      | 11.2.  | .1.1 Synopsis  | 11-8  |
|    |      | 11.2.  | .1.2 C++ Binding   | 11-8  |
|    |      | 11.2.  | 1.3 Argument   | 11-9  |
|    |      |        |  |       |

10.1.2 Performance Implications



10-3

| 11.2.1.4     | Exception               | 11-9  |
|--------------|-------------------------|-------|
| 11.2.1.5     | Description             | 11-9  |
| 11.2.1.6     | Return Value            | 11-9  |
| 11.2.2 start | _transient              | 11-9  |
| 11.2.2.1     | Synopsis                | 11-9  |
| 11.2.2.2     | IDL                     | 11-9  |
| 11.2.2.3     | C++ Binding             | 11-10 |
| 11.2.2.4     | Arguments               | 11-10 |
| 11.2.2.5     | Exceptions              | 11-10 |
| 11.2.2.6     | Description             | 11-10 |
| 11.2.2.7     | Return Value            | 11-10 |
| 11.2.3 start | _persistent_systemid    | 11-10 |
| 11.2.3.1     | Synopsis                | 11-11 |
| 11.2.3.2     | IDL                     | 11-11 |
| 11.2.3.3     | C++ Binding             | 11-11 |
| 11.2.3.4     | Arguments               | 11-11 |
| 11.2.3.5     | Exceptions              | 11-11 |
| 11.2.3.6     | Description             | 11-12 |
| 11.2.3.7     | Return Value            | 11-12 |
| 11.2.4 resta | art_persistent_systemid | 11-12 |
| 11.2.4.1     | Synopsis                | 11-12 |
| 11.2.4.2     | IDL                     | 11-12 |
| 11.2.4.3     | C++ Binding             | 11-13 |
| 11.2.4.4     | Arguments               | 11-13 |
| 11.2.4.5     | Exceptions              | 11-13 |
| 11.2.4.6     | Description             | 11-13 |
| 11.2.4.7     | Return Value            | 11-14 |
| 11.2.5 start | _persistent_userid      | 11-14 |
| 11.2.5.1     | Synopsis                | 11-14 |
| 11.2.5.2     | IDL                     | 11-14 |
| 11.2.5.3     | C++ Binding             | 11-14 |
| 11.2.5.4     | Arguments               | 11-14 |
| 11.2.5.5     | Exceptions              | 11-15 |
| 11.2.5.6     | Description             | 11-15 |
| 11.2.5.7     | Return Value            | 11-15 |
| 11.2.6 stop  | _object                 | 11-15 |
| 11.2.6.1     | Synopsis                | 11-16 |
| 11.2.6.2     | IDL                     | 11-16 |
| 11.2.6.3     | C++ Binding             | 11-16 |
| 11.2.6.4     | Argument                | 11-16 |
| 11.2.6.5     | Exceptions              | 11-16 |
| 11.2.6.6     | Description             | 11-16 |



| 11.2.6.7     | Return Value                                | 11-16 |
|--------------|---|-------|
| 11.2.7 stop  | o_all_objects                               | 11-16 |
| 11.2.7.1     | Synopsis                                    | 11-17 |
| 11.2.7.2     | IDL   | 11-17 |
| 11.2.7.3     | C++ Binding                                 | 11-17 |
| 11.2.7.4     | Exceptions                                  | 11-17 |
| 11.2.7.5     | Description                                 | 11-17 |
| 11.2.7.6     | Usage Note                                  | 11-17 |
| 11.2.7.7     | Return Value                                | 11-17 |
| 11.2.8 get_  | _string_oid                                 | 11-17 |
| 11.2.8.1     | Synopsis                                    | 11-17 |
| 11.2.8.2     | IDL   | 11-18 |
| 11.2.8.3     | C++ Binding                                 | 11-18 |
| 11.2.8.4     | JExceptions                                 | 11-18 |
| 11.2.8.5     | Description                                 | 11-18 |
| 11.2.8.6     | Return Value                                | 11-18 |
| 11.2.9 ~Ca   | ullbacks                                    | 11-18 |
| 11.2.9.1     | Synopsis                                    | 11-18 |
| 11.2.9.2     | C++ Binding                                 | 11-18 |
| 11.2.9.3     | Arguments                                   | 11-18 |
| 11.2.9.4     | Exceptions                                  | 11-19 |
| 11.2.9.5     | Description                                 | 11-19 |
| 11.2.9.6     | Usage Note                                  | 11-19 |
| 11.2.9.7     | Return Value                                | 11-19 |
| Developmer   | nt Commands                                 |       |
| Mapping of ( | OMG IDL Statements to C++                   | 13-1  |
| 0            | a Types                                     | 13-3  |
| 13.1.1.1     | Basic Data Types                            | 13-3  |
| 13.1.1.2     | Complex Data Types                          | 13-3  |
| 13.1.2 Strii |   | 13-4  |
| 13.1.3 wch   |   | 13-4  |
|              | rings                                       | 13-5  |
|              | nstants                                     | 13-5  |
| 13.1.6 Enu   |   | 13-6  |
| 13.1.7 Stru  |   | 13-7  |
|              | Fixed-length Versus Variable-length Structs | 13-7  |
|              |   |       |



| 13.1.   | .7.2 Member Mapping                                       | 13-8         |
|---------|---|--------------|
| 13.1.   | .7.3 Var  | 13-8         |
| 13.1.   | .7.4 Out  | 13-8         |
| 13.1.8  | Unions  | 13-8         |
| 13.1.   | .8.1 Union Member Accessor and Modifier Member Function M | Mapping 13-9 |
| 13.1.   | .8.2 Var  | 13-11        |
| 13.1.   | .8.3 Out  | 13-11        |
| 13.1.   | .8.4 Member Functions                                     | 13-11        |
| 13.1.9  | Sequences   | 13-12        |
| 13.1.   | .9.1 Sequence Element Mapping                             | 13-14        |
| 13.1.   | .9.2 Vars   | 13-14        |
| 13.1.   | .9.3 Out  | 13-14        |
| 13.1.   | .9.4 Member Functions                                     | 13-14        |
| 13.1.10 | Arrays  | 13-16        |
| 13.1.   | .10.1 Array Slice   | 13-16        |
| 13.1.   | .10.2 Array Element Mapping                               | 13-17        |
| 13.1.   | .10.3 Vars  | 13-17        |
| 13.1.   | .10.4 Out   | 13-17        |
| 13.1.   | .10.5 Allocation Member Functions                         | 13-17        |
| 13.1.11 | Exceptions  | 13-17        |
| 13.1.   | .11.1 Member Mapping                                      | 13-18        |
| 13.1.   | .11.2 Var   | 13-18        |
| 13.1.   | .11.3 Out   | 13-19        |
| 13.1.   | .11.4 Member Functions                                    | 13-19        |
| 13.1.12 | Mapping of Pseudo-objects to C++                          | 13-19        |
| 13.1.13 | Usage   | 13-20        |
| 13.1.14 | Mapping Rules   | 13-20        |
| 13.1.15 | Relation to the C PIDL Mapping                            | 13-21        |
| 13.1.16 | Typedefs  | 13-21        |
| 13.1.17 | Implementing Interfaces                                   | 13-23        |
| 13.1.   | .17.1 Argument Mapping                                    | 13-24        |
| 13.1.18 | Implementing Operations                                   | 13-24        |
| 13.1.   | .18.1 Skeleton Derivation from Object                     | 13-25        |
| 13.1.19 | PortableServer Functions                                  | 13-25        |
| 13.1.20 | Modules   | 13-26        |
| 13.1.21 | Interfaces  | 13-27        |
| 13.1.22 | Generated Static Member Functions                         | 13-28        |
| 13.1.23 | Object Reference Types                                    | 13-28        |
| 13.1.24 | Attributes  | 13-28        |
| 13.1.   | .24.1 Argument Mapping                                    | 13-30        |
| 13.1.25 | Any Type  | 13-30        |
| 13.1.   | .25.1 Handling Typed Values                               | 13-30        |



|     | 13.1.25.2 Insertion into Any                                       | 13-31 |
|-----|--|-------|
|     | 13.1.25.3 Extraction from Any                                      | 13-33 |
|     | 13.1.25.4 Distinguishing Boolean, Octet, Char, and Bounded Strings | 13-35 |
|     | 13.1.25.5 Widening to Object                                       | 13-37 |
|     | 13.1.25.6 Handling Untyped Values                                  | 13-37 |
|     | 13.1.25.7 Any Constructors, Destructor, Assignment Operator        | 13-38 |
|     | 13.1.25.8 The Any Class  | 13-39 |
|     | 13.1.26 Value Type   | 13-39 |
|     | 13.1.26.1 Overview   | 13-39 |
|     | 13.1.26.2 Architecture   | 13-39 |
|     | 13.1.26.3 Benefits   | 13-40 |
|     | 13.1.26.4 Valuetype Example  | 13-40 |
|     | 13.2 Fixed-length Versus Variable-length User-defined Types        | 13-41 |
|     | 13.3 Using var Classes   | 13-42 |
|     | 13.3.1 Sequence vars   | 13-44 |
|     | 13.3.2 Array vars  | 13-44 |
|     | 13.3.3 String vars   | 13-45 |
|     | 13.3.3.1 out Classes   | 13-45 |
|     | 13.4 Using out Classes   | 13-46 |
|     | 13.4.1 Object Reference out Parameter                              | 13-48 |
|     | 13.4.2 Sequence outs   | 13-49 |
|     | 13.4.3 Array outs  | 13-49 |
|     | 13.4.4 String outs   | 13-49 |
|     | 13.5 Argument Passing Considerations                               | 13-51 |
|     | 13.5.1 Operation Parameters and Signatures                         | 13-53 |
| 4.4 |  |       |
| 14  | CORBA API  |       |
|     | 14.1 Global Classes  | 14-2  |
|     | 14.2 Pseudo-objects  | 14-2  |
|     | 14.3 Any Class Member Functions                                    | 14-2  |
|     | 14.3.1 CORBA::Any::Any()   | 14-3  |
|     | 14.3.1.1 Synopsis  | 14-4  |
|     | 14.3.1.2 C++ Binding   | 14-4  |
|     | 14.3.1.3 Arguments   | 14-4  |
|     | 14.3.1.4 Description   | 14-4  |
|     | 14.3.1.5 Return Values   | 14-4  |
|     | 14.3.2 CORBA::Any::Any(const CORBA::Any & InitAny)                 | 14-4  |
|     | 14.3.2.1 Synopsis  | 14-4  |
|     | 14.3.2.2 C++ Binding   | 14-4  |
|     | 14.3.2.3 Argument  | 14-4  |
|     | 14.3.2.4 Description   | 14-5  |
|     |  |       |



| 14.    | 3.2.5 | Return Values   | 14-5  |
|--------|-------|---|-------|
| 14.3.3 | COR   | RBA::Any::Any(TypeCode_ptr TC, void * Value, Boolean Release) | 14-5  |
| 14.    | 3.3.1 | Synopsis  | 14-5  |
| 14.    | 3.3.2 | C++ Binding   | 14-5  |
| 14.    | 3.3.3 | Arguments   | 14-5  |
| 14.    | 3.3.4 | Description   | 14-5  |
| 14.    | 3.3.5 | Return Values   | 14-6  |
| 14.3.4 | COR   | RBA::Any::~Any()  | 14-6  |
| 14.    | 3.4.1 | Synopsis  | 14-6  |
| 14.    | 3.4.2 | C++ Binding   | 14-6  |
| 14.    | 3.4.3 | Arguments   | 14-6  |
| 14.    | 3.4.4 | Description   | 14-6  |
| 14.    | 3.4.5 | Return Values   | 14-6  |
| 14.3.5 | COR   | RBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)  | 14-6  |
| 14.    | 3.5.1 | Synopsis  | 14-6  |
| 14.    | 3.5.2 | C++ Binding   | 14-7  |
| 14.    | 3.5.3 | Arguments   | 14-7  |
| 14.    | 3.5.4 | Description   | 14-7  |
| 14.    | 3.5.5 | Return Values   | 14-7  |
| 14.3.6 | void  | CORBA::any::operator<<=()                                     | 14-7  |
| 14.    | 3.6.1 | Synopsis  | 14-7  |
| 14.    | 3.6.2 | C++ Binding   | 14-7  |
| 14.    | 3.6.3 | Argument  | 14-8  |
| 14.    | 3.6.4 | Description   | 14-8  |
| 14.    | 3.6.5 | Return Values   | 14-8  |
| 14.3.7 | COR   | RBA::Boolean CORBA::Any::operator>>=()                        | 14-8  |
| 14.    | 3.7.1 | Synopsis  | 14-8  |
| 14.    | 3.7.2 | C++ Binding   | 14-8  |
| 14.    | 3.7.3 | Argument  | 14-9  |
| 14.    | 3.7.4 | Description   | 14-9  |
| 14.    | 3.7.5 | Return Values   | 14-9  |
| 14.3.8 | COR   | RBA::Any::operator<<=()                                       | 14-9  |
| 14.    | 3.8.1 | Synopsis  | 14-9  |
| 14.    | 3.8.2 | C++ Binding   | 14-9  |
| 14.    | 3.8.3 | Argument  | 14-9  |
| 14.    | 3.8.4 | Description   | 14-9  |
| 14.    | 3.8.5 | Return Values   | 14-10 |
| 14.3.9 | COR   | RBA::Boolean CORBA::Any::operator>>=()                        | 14-10 |
| 14.3   | 3.9.1 | Synopsis  | 14-10 |
| 14.3   | 3.9.2 | C++ Binding   | 14-10 |
| 14.    | 3.9.3 | Argument  | 14-10 |
| 14.    | 3.9.4 | Description   | 14-10 |
|        |       |   |       |



| 14.3.9.5        | Return Values                              | 14-10 |
|-----------------|--|-------|
| 14.3.10 CO      | RBA::TypeCode_ptr CORBA::Any::type() const | 14-10 |
| 14.3.10.1       | Synopsis                                   | 14-11 |
| 14.3.10.2       | C++ Binding                                | 14-11 |
| 14.3.10.3       | Arguments                                  | 14-11 |
| 14.3.10.4       | Description                                | 14-11 |
| 14.3.10.5       | Return Values                              | 14-11 |
| 14.3.11 void    | d CORBA::Any::replace()                    | 14-11 |
| 14.3.11.1       | Synopsis                                   | 14-11 |
| 14.3.11.2       | C++ Binding                                | 14-11 |
| 14.3.11.3       | Arguments                                  | 14-12 |
| 14.3.11.4       | Description                                | 14-12 |
| 14.3.11.5       | Return Values                              | 14-12 |
| 14.4 Context M  | ember Functions                            | 14-12 |
| 14.4.1 Mem      | nory Management                            | 14-13 |
| 14.4.1.1        | CORBA::Context::context_name               | 14-13 |
| 14.4.1.2        | CORBA::Context::create_child               | 14-14 |
| 14.4.1.3        | CORBA::Context::delete_values              | 14-15 |
| 14.4.1.4        | CORBA::Context::get_values                 | 14-16 |
| 14.4.1.5        | CORBA::Context::parent                     | 14-17 |
| 14.4.1.6        | CORBA::Context::set_one_value              | 14-18 |
| 14.4.1.7        | CORBA::Context::set_values                 | 14-19 |
| 14.5 ContextLis | st Member Functions                        | 14-20 |
| 14.5.1 COF      | RBA::ContextList:: count                   | 14-20 |
| 14.5.1.1        | Synopsis                                   | 14-21 |
| 14.5.1.2        | C++ Binding                                | 14-21 |
| 14.5.1.3        | Arguments                                  | 14-21 |
| 14.5.1.4        | Exception                                  | 14-21 |
| 14.5.1.5        | Description                                | 14-21 |
| 14.5.1.6        | Return Values                              | 14-21 |
| 14.5.1.7        | See Also                                   | 14-21 |
| 14.5.2 COF      | RBA::ContextList::add                      | 14-21 |
| 14.5.2.1        | Synopsis                                   | 14-21 |
| 14.5.2.2        | C++ Binding                                | 14-22 |
| 14.5.2.3        | Argument                                   | 14-22 |
| 14.5.2.4        | Exception                                  | 14-22 |
| 14.5.2.5        | Description                                | 14-22 |
| 14.5.2.6        | Return Values                              | 14-22 |
| 14.5.2.7        | See Also                                   | 14-22 |
| 14.5.3 COF      | RBA::ContextList::add_consume              | 14-22 |
| 14.5.3.1        | Synopsis                                   | 14-22 |
| 14.5.3.2        | C++ Binding                                | 14-23 |



| 14.5.3      | 3.3 Argument                   | 14-23 |
|-------------|--------------------------------|-------|
| 14.5.3      | 3.4 Exception                  | 14-23 |
| 14.5.3      | 3.5 Description                | 14-23 |
| 14.5.3      | 3.6 Return Values              | 14-23 |
| 14.5.3      | 3.7 See Also                   | 14-23 |
| 14.5.4      | CORBA::ContextList::item       | 14-23 |
| 14.5.4      | 4.1 Synopsis                   | 14-23 |
| 14.5.4      | 4.2 C++ Binding                | 14-23 |
| 14.5.4      | 4.3 Argument                   | 14-24 |
| 14.5.4      | 4.4 Exceptions                 | 14-24 |
| 14.5.4      | 4.5 Description                | 14-24 |
| 14.5.4      | 4.6 Return Values              | 14-24 |
| 14.5.4      | 4.7 See Also                   | 14-24 |
| 14.5.5      | CORBA::ContextList::remove     | 14-24 |
| 14.5.       | 5.1 Synopsis                   | 14-24 |
| 14.5.       | 5.2 C++ Binding                | 14-24 |
| 14.5.       | 5.3 Argument                   | 14-25 |
| 14.5.       | 5.4 Exceptions                 | 14-25 |
| 14.5.       | 5.5 Description                | 14-25 |
| 14.5.       | 5.6 Return Values              | 14-25 |
| 14.5.       | 5.7 See Also                   | 14-25 |
| 14.6 Name   | dValue Member Functions        | 14-25 |
| 14.6.1      | Memory Management              | 14-25 |
| 14.6.2      | 1.1 CORBA::NamedValue::flags   | 14-26 |
| 14.6.2      | 1.2 CORBA::NamedValue::name    | 14-26 |
| 14.6.2      | 1.3 CORBA::NamedValue::value   | 14-27 |
| 14.7 NVLis  | t Member Functions             | 14-28 |
| 14.7.1      | Memory Management              | 14-28 |
| 14.7.2      | 1.1 CORBA::NVList::add         | 14-29 |
| 14.7.2      | 1.2 CORBA::NVList::add_item    | 14-30 |
| 14.7.2      | 1.3 CORBA::NVList::add_value   | 14-31 |
| 14.7.2      | 1.4 CORBA::NVList::count       | 14-32 |
| 14.7.2      | 1.5 CORBA::NVList::item        | 14-33 |
| 14.7.2      | 1.6 CORBA::NVList::remove      | 14-34 |
| 14.8 Object | t Member Functions             | 14-35 |
| 14.8.1      | CORBA::Object::_create_request | 14-36 |
| 14.8.2      | 1.1 Synopsis                   | 14-36 |
| 14.8.2      | 1.2 C++ Binding                | 14-36 |
| 14.8.2      | 1.3 Arguments                  | 14-37 |
| 14.8.2      | 1.4 Description                | 14-37 |
| 14.8.2      | 1.5 Return Values              | 14-37 |
| 14.8.2      | 1.6 See Also                   | 14-37 |



| 14.8 | 3.2  | COR  | BA::Object::_duplicate     | 14-37 |
|------|------|------|----------------------------|-------|
|      | 14.8 | 2.1  | Synopsis                   | 14-38 |
|      | 14.8 | 2.2  | C++ Binding                | 14-38 |
|      | 14.8 | 2.3  | Argument                   | 14-38 |
|      | 14.8 | 2.4  | Description                | 14-38 |
|      | 14.8 | 2.5  | Return Values              | 14-38 |
|      | 14.8 | 2.6  | Example                    | 14-38 |
| 14.8 | 3.3  | COR  | BA::Object::_get_interface | 14-38 |
|      | 14.8 | 3.1  | Synopsis                   | 14-38 |
|      | 14.8 | 3.2  | C++ Binding                | 14-39 |
|      | 14.8 | .3.3 | Arguments                  | 14-39 |
|      | 14.8 | 3.4  | Description                | 14-39 |
|      | 14.8 | 3.5  | Return Values              | 14-39 |
| 14.8 | 3.4  | COR  | BA::Object::_is_a          | 14-39 |
|      | 14.8 | 4.1  | Synopsis                   | 14-39 |
|      | 14.8 | 4.2  | C++ Binding                | 14-39 |
|      | 14.8 | .4.3 | Argument                   | 14-39 |
|      | 14.8 | 4.4  | Description                | 14-40 |
|      | 14.8 | .4.5 | Return Values              | 14-40 |
|      | 14.8 | 4.6  | Example                    | 14-40 |
| 14.8 | 3.5  | COR  | BA::Object::_is_equivalent | 14-40 |
|      | 14.8 | 5.1  | Synopsis                   | 14-40 |
|      | 14.8 | 5.2  | C++ Binding                | 14-40 |
|      | 14.8 | 5.3  | Argument                   | 14-40 |
|      | 14.8 | 5.4  | Exceptions                 | 14-40 |
|      | 14.8 | 5.5  | Description                | 14-41 |
|      | 14.8 | 5.6  | Return Values              | 14-41 |
|      | 14.8 | 5.7  | Example                    | 14-41 |
| 14.8 | 3.6  | COR  | BA::Object::_nil           | 14-41 |
|      | 14.8 | 6.1  | Synopsis                   | 14-41 |
|      | 14.8 | 6.2  | C++ Binding                | 14-41 |
|      | 14.8 | .6.3 | Arguments                  | 14-41 |
|      | 14.8 | 6.4  | Description                | 14-41 |
|      | 14.8 | 6.5  | Return Values              | 14-42 |
|      | 14.8 | 6.6  | Example                    | 14-42 |
| 14.8 | 3.7  | COR  | BA::Object::_non_existent  | 14-42 |
|      | 14.8 | 7.1  | Synopsis                   | 14-42 |
|      | 14.8 | 7.2  | C++ Binding                | 14-42 |
|      | 14.8 | 7.3  | Arguments                  | 14-42 |
|      | 14.8 | 7.4  | Description                | 14-42 |
|      | 14.8 | 7.5  | Return Values              | 14-42 |
| 14.8 | 8.8  | COR  | BA::Object::_request       | 14-42 |



|       | 14.8.8.1  | Synopsis                        | 14-43 |
|-------|-----------|---------------------------------|-------|
|       | 14.8.8.2  | C++ Binding                     | 14-43 |
|       | 14.8.8.3  | Argument                        | 14-43 |
|       | 14.8.8.4  | Description                     | 14-43 |
|       | 14.8.8.5  | Return Values                   | 14-43 |
|       | 14.8.8.6  | See Also                        | 14-43 |
| 14.9  | CORBA N   | Member Functions                | 14-43 |
| 14    | 1.9.1 CO  | RBA::release                    | 14-44 |
|       | 14.9.1.1  | Synopsis                        | 14-44 |
|       | 14.9.1.2  | C++ Binding                     | 14-44 |
|       | 14.9.1.3  | Argument                        | 14-44 |
|       | 14.9.1.4  | Description                     | 14-44 |
|       | 14.9.1.5  | Return Values                   | 14-45 |
|       | 14.9.1.6  | Example                         | 14-45 |
| 14    | 1.9.2 CO  | RBA::is_nil                     | 14-45 |
|       | 14.9.2.1  | Synopsis                        | 14-45 |
|       | 14.9.2.2  | C++ Binding                     | 14-45 |
|       | 14.9.2.3  | Argument                        | 14-45 |
|       | 14.9.2.4  | Description                     | 14-45 |
|       | 14.9.2.5  | Return Values                   | 14-45 |
|       | 14.9.2.6  | Example                         | 14-46 |
| 14    | 1.9.3 CO  | RBA::hash                       | 14-46 |
|       | 14.9.3.1  | Synopsis                        | 14-46 |
|       | 14.9.3.2  | C++ Binding                     | 14-46 |
|       | 14.9.3.3  | Argument                        | 14-46 |
|       | 14.9.3.4  | Description                     | 14-46 |
|       | 14.9.3.5  | Return Values                   | 14-46 |
| 14    | 1.9.4 CO  | RBA::resolve_initial_references | 14-47 |
|       | 14.9.4.1  | Synopsis                        | 14-47 |
|       | 14.9.4.2  | C++ Binding                     | 14-47 |
|       | 14.9.4.3  | Argument                        | 14-47 |
|       | 14.9.4.4  | Exception                       | 14-47 |
|       | 14.9.4.5  | Description                     | 14-47 |
|       | 14.9.4.6  | Return Values                   | 14-47 |
|       | 14.9.4.7  | Example                         | 14-47 |
| 14.10 | ORB Me    | ember Functions                 | 14-48 |
| 14    | 1.10.1 CC | DRBA::ORB::clear_ctx            | 14-49 |
|       | 14.10.1.1 | L Synopsis                      | 14-49 |
|       | 14.10.1.2 | 2 C++ Binding                   | 14-49 |
|       | 14.10.1.3 | B Parameters                    | 14-50 |
|       | 14.10.1.4 | 4 Return Value                  | 14-50 |
|       | 14.10.1.5 | 5 Description                   | 14-50 |



| 14.10.1.6   | Example                         | 14-50 |
|-------------|---------------------------------|-------|
| 14.10.1.7   | See Also                        | 14-50 |
| 14.10.2 COF | RBA::ORB::create_context_list   | 14-50 |
| 14.10.2.1   | Synopsis                        | 14-50 |
| 14.10.2.2   | C++ Binding                     | 14-50 |
| 14.10.2.3   | Argument                        | 14-51 |
| 14.10.2.4   | Description                     | 14-51 |
| 14.10.2.5   | Return Values                   | 14-51 |
| 14.10.3 COF | RBA::ORB::create_environment    | 14-51 |
| 14.10.3.1   | Synopsis                        | 14-51 |
| 14.10.3.2   | C++ Binding                     | 14-51 |
| 14.10.3.3   | Argument                        | 14-51 |
| 14.10.3.4   | Description                     | 14-51 |
| 14.10.3.5   | Return Values                   | 14-51 |
| 14.10.3.6   | See Also                        | 14-52 |
| 14.10.4 COF | RBA::ORB::create_exception_list | 14-52 |
| 14.10.4.1   | Synopsis                        | 14-52 |
| 14.10.4.2   | C++ Binding                     | 14-52 |
| 14.10.4.3   | Argument                        | 14-52 |
| 14.10.4.4   | Description                     | 14-52 |
| 14.10.4.5   | Return Values                   | 14-52 |
| 14.10.5 COF | RBA::ORB::create_list           | 14-52 |
| 14.10.5.1   | Synopsis                        | 14-53 |
| 14.10.5.2   | C++ Binding                     | 14-53 |
| 14.10.5.3   | Arguments                       | 14-53 |
| 14.10.5.4   | Description                     | 14-53 |
| 14.10.5.5   | Return Values                   | 14-53 |
| 14.10.5.6   | See Also                        | 14-53 |
| 14.10.6 COF | RBA::ORB::create_named_value    | 14-53 |
| 14.10.6.1   | Synopsis                        | 14-53 |
| 14.10.6.2   | C++ Binding                     | 14-54 |
| 14.10.6.3   | Argument                        | 14-54 |
| 14.10.6.4   | Description                     | 14-54 |
| 14.10.6.5   | Return Values                   | 14-54 |
| 14.10.6.6   | See Also                        | 14-54 |
| 14.10.7 COF | RBA::ORB::create_operation_list | 14-54 |
| 14.10.7.1   | Synopsis                        | 14-54 |
| 14.10.7.2   | C++ Binding                     | 14-55 |
| 14.10.7.3   | Arguments                       | 14-55 |
| 14.10.7.4   | Description                     | 14-55 |
| 14.10.7.5   | Return Values                   | 14-55 |
| 14.10.7.6   | See Also                        | 14-55 |



| 14.10.8 COR  | BA::ORB::create_policy        | 14-55 |
|--------------|-------------------------------|-------|
| 14.10.8.1    | Synopsis                      | 14-55 |
| 14.10.8.2    | C++ Binding                   | 14-56 |
| 14.10.8.3    | Arguments                     | 14-56 |
| 14.10.8.4    | Exceptions                    | 14-56 |
| 14.10.8.5    | Description                   | 14-56 |
| 14.10.8.6    | Return Values                 | 14-57 |
| 14.10.8.7    | Example                       | 14-57 |
| 14.10.9 COR  | BA::ORB::destroy              | 14-58 |
| 14.10.9.1    | Synopsis                      | 14-58 |
| 14.10.9.2    | C++ Binding                   | 14-58 |
| 14.10.9.3    | Parameter                     | 14-58 |
| 14.10.9.4    | Return Value                  | 14-58 |
| 14.10.9.5    | Description                   | 14-58 |
| 14.10.9.6    | Example                       | 14-58 |
| 14.10.10 COF | RBA::ORB::get_ctx             | 14-58 |
| 14.10.10.1   | Synopsis                      | 14-59 |
| 14.10.10.2   | C++ Binding                   | 14-59 |
| 14.10.10.3   | Arguments                     | 14-59 |
| 14.10.10.4   | Return Value                  | 14-59 |
| 14.10.10.5   | Description                   | 14-59 |
| 14.10.10.6   | Example                       | 14-59 |
| 14.10.10.7   | See Also                      | 14-59 |
| 14.10.11 COF | RBA::ORB::get_default_context | 14-59 |
| 14.10.11.1   | Synopsis                      | 14-60 |
| 14.10.11.2   | C++ Binding                   | 14-60 |
| 14.10.11.3   | Argument                      | 14-60 |
| 14.10.11.4   | Description                   | 14-60 |
| 14.10.11.5   | Return Values                 | 14-60 |
| 14.10.11.6   | See Also                      | 14-60 |
| 14.10.12 COF | RBA::ORB::get_next_response   | 14-60 |
| 14.10.12.1   | Synopsis                      | 14-60 |
| 14.10.12.2   | C++ Binding                   | 14-60 |
| 14.10.12.3   | Argument                      | 14-61 |
| 14.10.12.4   | Description                   | 14-61 |
| 14.10.12.5   | Return Values                 | 14-61 |
| 14.10.12.6   | See Also                      | 14-61 |
| 14.10.13 COF | RBA::ORB::inform_thread_exit  | 14-61 |
| 14.10.13.1   | Synopsis                      | 14-61 |
| 14.10.13.2   | C++ Binding                   | 14-61 |
| 14.10.13.3   | Parameter                     | 14-61 |
| 14.10.13.4   | Return Value                  | 14-62 |



| 14.10.13.5   | Description                               | 14-62 |
|--------------|---|-------|
| 14.10.13.6   | Example                                   | 14-62 |
| 14.10.14 COF | RBA::ORB::list_initial_services           | 14-62 |
| 14.10.14.1   | Synopsis                                  | 14-62 |
| 14.10.14.2   | C++ Binding                               | 14-62 |
| 14.10.14.3   | Argument                                  | 14-62 |
| 14.10.14.4   | Description                               | 14-63 |
| 14.10.14.5   | Return Values                             | 14-63 |
| 14.10.14.6   | See Also                                  | 14-63 |
| 14.10.15 COF | RBA::ORB::object_to_string                | 14-63 |
| 14.10.15.1   | Synopsis                                  | 14-63 |
| 14.10.15.2   | C++ Binding                               | 14-63 |
| 14.10.15.3   | Argument                                  | 14-63 |
| 14.10.15.4   | Description                               | 14-64 |
| 14.10.15.5   | Return Values                             | 14-64 |
| 14.10.15.6   | Example                                   | 14-64 |
| 14.10.15.7   | See Also                                  | 14-64 |
| 14.10.16 COF | RBA::ORB::perform_work                    | 14-64 |
| 14.10.16.1   | Synopsis                                  | 14-64 |
| 14.10.16.2   | C++ Binding                               | 14-64 |
| 14.10.16.3   | Arguments                                 | 14-64 |
| 14.10.16.4   | Exceptions                                | 14-65 |
| 14.10.16.5   | Description                               | 14-65 |
| 14.10.16.6   | Return Values                             | 14-65 |
| 14.10.16.7   | See Also                                  | 14-65 |
| 14.10.16.8   | Example                                   | 14-65 |
| 14.10.17 COF | RBA::ORB::poll_next_response              | 14-65 |
| 14.10.17.1   | Synopsis                                  | 14-66 |
| 14.10.17.2   | C++ Binding                               | 14-66 |
| 14.10.17.3   | Arguments                                 | 14-66 |
| 14.10.17.4   | Description                               | 14-66 |
| 14.10.17.5   | Return Values                             | 14-66 |
| 14.10.17.6   | See Also                                  | 14-66 |
| 14.10.18 COF | RBA::ORB::resolve_initial_references      | 14-66 |
| 14.10.18.1   | Synopsis                                  | 14-66 |
| 14.10.18.2   | C++ Binding                               | 14-66 |
| 14.10.18.3   | Augument                                  | 14-67 |
| 14.10.18.4   | Description                               | 14-67 |
| 14.10.18.5   | Return Values                             | 14-67 |
| 14.10.18.6   | See Also                                  | 14-67 |
| 14.10.19 COF | RBA::ORB::send_multiple_requests_deferred | 14-67 |
| 14.10.19.1   | Synopsis                                  | 14-67 |



|       | 14.10.19.2  | C++ Binding                             | 14-68 |
|-------|-------------|---|-------|
|       | 14.10.19.3  | Argument                                | 14-68 |
|       | 14.10.19.4  | Description                             | 14-68 |
|       | 14.10.19.5  | Return Values                           | 14-68 |
|       | 14.10.19.6  | See Also                                | 14-68 |
| 14    | .10.20 CO   | RBA::ORB::send_multiple_requests_oneway | 14-68 |
|       | 14.10.20.1  | Synopsis                                | 14-68 |
|       | 14.10.20.2  | C++ Binding                             | 14-68 |
|       | 14.10.20.3  | Argument                                | 14-69 |
|       | 14.10.20.4  | Description                             | 14-69 |
|       | 14.10.20.5  | Return Values                           | 14-69 |
|       | 14.10.20.6  | See Also                                | 14-69 |
| 14    | .10.21 CO   | RBA::ORB::set_ctx                       | 14-69 |
|       | 14.10.21.1  | Synopsis                                | 14-69 |
|       | 14.10.21.2  | C++ Binding                             | 14-69 |
|       | 14.10.21.3  | Parameter                               | 14-69 |
|       | 14.10.21.4  | Return Value                            | 14-69 |
|       | 14.10.21.5  | Description                             | 14-70 |
|       | 14.10.21.6  | Example                                 | 14-70 |
|       | 14.10.21.7  | See Also                                | 14-70 |
| 14    | .10.22 CO   | RBA::ORB::string_to_object              | 14-70 |
|       | 14.10.22.1  | Synopsis                                | 14-70 |
|       | 14.10.22.2  | C++ Binding                             | 14-70 |
|       | 14.10.22.3  | Argument                                | 14-71 |
|       | 14.10.22.4  | Description                             | 14-71 |
|       | 14.10.22.5  | Return Value                            | 14-71 |
|       | 14.10.22.6  | See Also                                | 14-71 |
| 14    | .10.23 CO   | RBA::ORB::work_pending                  | 14-71 |
|       | 14.10.23.1  | Synopsis                                | 14-71 |
|       | 14.10.23.2  | C++ Binding                             | 14-72 |
|       | 14.10.23.3  | Arguments                               | 14-72 |
|       | 14.10.23.4  | Description                             | 14-72 |
|       | 14.10.23.5  | Return Values                           | 14-72 |
|       | 14.10.23.6  | See Also                                | 14-72 |
| 14.11 | ORB Initial | ization Member Function                 | 14-72 |
| 14    | .11.1 COR   | BA::ORB_init                            | 14-72 |
|       | 14.11.1.1   | Synopsis                                | 14-73 |
|       | 14.11.1.2   | C++ Binding                             | 14-73 |
|       | 14.11.1.3   | Arguments                               | 14-73 |
|       | 14.11.1.4   | Description                             | 14-73 |
|       | 14.11.1.5   | Return Value                            | 14-74 |
|       | 14.11.1.6   | Exceptions                              | 14-74 |



| 14    | .11.2 ORB   |  | 14-75 |
|-------|-------------|--|-------|
|       | 14.11.2.1   | Synopsis                                     | 14-75 |
|       | 14.11.2.2   | Syntax                                       | 14-75 |
|       | 14.11.2.3   | Description                                  | 14-75 |
|       | 14.11.2.4   | Parameters                                   | 14-75 |
|       | 14.11.2.5   | Portability                                  | 14-78 |
|       | 14.11.2.6   | Interoperability                             | 14-78 |
|       | 14.11.2.7   | Examples                                     | 14-78 |
|       | 14.11.2.8   | See Also                                     | 14-79 |
| 14.12 | Policy Mer  | nber Functions                               | 14-79 |
| 14    | .12.1 COR   | RBA:Policy::copy                             | 14-80 |
|       | 14.12.1.1   | Synopsis                                     | 14-80 |
|       | 14.12.1.2   | C++ Binding                                  | 14-80 |
|       | 14.12.1.3   | Argument                                     | 14-80 |
|       | 14.12.1.4   | Description                                  | 14-80 |
|       | 14.12.1.5   | Return Values                                | 14-80 |
| 14    | .12.2 COR   | RBA::Policy::destroy                         | 14-80 |
|       | 14.12.2.1   | Synopsis                                     | 14-81 |
|       | 14.12.2.2   | C++ Binding                                  | 14-81 |
|       | 14.12.2.3   | Argument                                     | 14-81 |
|       | 14.12.2.4   | Exceptions                                   | 14-81 |
|       | 14.12.2.5   | Description                                  | 14-81 |
|       | 14.12.2.6   | Return Values                                | 14-81 |
| 14.13 | PortableSe  | erver Member Functions                       | 14-81 |
| 14    | .13.1 Porta | ableServer::POA::activate_object             | 14-82 |
|       | 14.13.1.1   | Synopsis                                     | 14-82 |
|       | 14.13.1.2   | C++ Binding                                  | 14-82 |
|       | 14.13.1.3   | Argument                                     | 14-83 |
|       | 14.13.1.4   | Exceptions                                   | 14-83 |
|       | 14.13.1.5   | Description                                  | 14-83 |
|       | 14.13.1.6   | Return Values                                | 14-83 |
|       | 14.13.1.7   | Example                                      | 14-83 |
| 14    | .13.2 Porta | ableServer::POA::activate_object_with_id     | 14-83 |
|       | 14.13.2.1   | Synopsis                                     | 14-84 |
|       | 14.13.2.2   | C++ Binding                                  | 14-84 |
|       | 14.13.2.3   | Argument                                     | 14-84 |
|       | 14.13.2.4   | Exceptions                                   | 14-84 |
|       | 14.13.2.5   | Description                                  | 14-84 |
|       | 14.13.2.6   | Return Values                                | 14-85 |
|       | 14.13.2.7   | Example                                      | 14-85 |
| 14    | .13.3 Porta | ableServer::POA::create_id_assignment_policy | 14-85 |
|       | 14.13.3.1   | Synopsis                                     | 14-85 |
|       |             |  |       |



| 14.13.3.2    | C++ Binding                                | 14-85 |
|--------------|--|-------|
| 14.13.3.3    | Argument                                   | 14-85 |
| 14.13.3.4    | Description                                | 14-85 |
| 14.13.3.5    | Return Values                              | 14-86 |
| 14.13.4 Port | tableServer::POA::create_lifespan_policy   | 14-86 |
| 14.13.4.1    | Synopsis                                   | 14-86 |
| 14.13.4.2    | C++ Binding                                | 14-86 |
| 14.13.4.3    | Argument                                   | 14-86 |
| 14.13.4.4    | Description                                | 14-86 |
| 14.13.4.5    | Return Values                              | 14-87 |
| 14.13.5 Port | tableServer::POA::create_POA               | 14-87 |
| 14.13.5.1    | Synopsis                                   | 14-87 |
| 14.13.5.2    | C++ Binding                                | 14-87 |
| 14.13.5.3    | Arguments                                  | 14-88 |
| 14.13.5.4    | Exceptions                                 | 14-88 |
| 14.13.5.5    | Description                                | 14-88 |
| 14.13.5.6    | Return Values                              | 14-88 |
| 14.13.5.7    | Examples                                   | 14-89 |
| 14.13.6 Port | tableServer::POA::create_reference         | 14-89 |
| 14.13.6.1    | Synopsis                                   | 14-89 |
| 14.13.6.2    | C++ Binding                                | 14-89 |
| 14.13.6.3    | Argument                                   | 14-90 |
| 14.13.6.4    | Exceptions                                 | 14-90 |
| 14.13.6.5    | Description                                | 14-90 |
| 14.13.6.6    | Return Values                              | 14-90 |
| 14.13.7 Port | tableServer::POA::create_reference_with_id | 14-90 |
| 14.13.7.1    | Synopsis                                   | 14-90 |
| 14.13.7.2    | C++ Binding                                | 14-90 |
| 14.13.7.3    | Arguments                                  | 14-91 |
| 14.13.7.4    | Exceptions                                 | 14-91 |
| 14.13.7.5    | Description                                | 14-91 |
| 14.13.7.6    | Return Values                              | 14-91 |
| 14.13.7.7    | Example                                    | 14-91 |
| 14.13.8 Port | tableServer::POA::deactivate_object        | 14-91 |
| 14.13.8.1    | Synopsis                                   | 14-92 |
| 14.13.8.2    | C++ Binding                                | 14-92 |
| 14.13.8.3    | Argument                                   | 14-92 |
| 14.13.8.4    | Exceptions                                 | 14-92 |
| 14.13.8.5    | Description                                | 14-92 |
| 14.13.8.6    | Return Values                              | 14-92 |
| 14.13.9 Port | tableServer::POA::destroy                  | 14-92 |
| 14.13.9.1    | Synopsis                                   | 14-92 |



| 14.13.9.2       | C++ Binding                            | 14-93 |
|-----------------|--|-------|
| 14.13.9.3       | Arguments                              | 14-93 |
| 14.13.9.4       | Description                            | 14-93 |
| 14.13.9.5       | Return Values                          | 14-93 |
| 14.13.10 Por    | tableServer::POA::find_POA             | 14-93 |
| 14.13.10.1      | Synopsis                               | 14-94 |
| 14.13.10.2      | C++ Binding                            | 14-94 |
| 14.13.10.3      | Argument                               | 14-94 |
| 14.13.10.4      | Exception                              | 14-94 |
| 14.13.10.5      | Description                            | 14-94 |
| 14.13.10.6      | Return Values                          | 14-94 |
| 14.13.11 Port   | tableServer::POA::reference_to_id      | 14-94 |
| 14.13.11.1      | Synopsis                               | 14-94 |
| 14.13.11.2      | C++ Binding                            | 14-94 |
| 14.13.11.3      | Argument                               | 14-95 |
| 14.13.11.4      | Exceptions                             | 14-95 |
| 14.13.11.5      | Description                            | 14-95 |
| 14.13.11.6      | Return Values                          | 14-95 |
| 14.13.12 Por    | tableServer::POA::the_POAManager       | 14-95 |
| 14.13.12.1      | Synopsis                               | 14-95 |
| 14.13.12.2      | C++ Binding                            | 14-95 |
| 14.13.12.3      | Argument                               | 14-95 |
| 14.13.12.4      | Description                            | 14-96 |
| 14.13.12.5      | Return Values                          | 14-96 |
| 14.13.12.6      | Example                                | 14-96 |
| 14.13.13 Por    | tableServer::ServantBase::_default_POA | 14-96 |
| 14.13.13.1      | Synopsis                               | 14-96 |
| 14.13.13.2      | C++ Binding                            | 14-96 |
| 14.13.13.3      | Argument                               | 14-96 |
| 14.13.13.4      | Description                            | 14-97 |
| 14.13.13.5      | Return Values                          | 14-97 |
| 14.14 POA Curre | ent Member Functions                   | 14-97 |
| 14.14.1 Porta   | ableServer::Current::get_object_id     | 14-97 |
| 14.14.1.1       | Synopsis                               | 14-97 |
| 14.14.1.2       | C++ Binding                            | 14-97 |
| 14.14.1.3       | Arguments                              | 14-97 |
| 14.14.1.4       | Exception                              | 14-98 |
| 14.14.1.5       | Description                            | 14-98 |
| 14.14.1.6       | Return Values                          | 14-98 |
| 14.14.2 Porta   | ableServer::Current::get_POA           | 14-98 |
| 14.14.2.1       | Synopsis                               | 14-98 |
| 14.14.2.2       | C++ Binding                            | 14-98 |



|       | 14.14.2.3   | Argument                           | 14-98  |
|-------|-------------|------------------------------------|--------|
|       | 14.14.2.4   | Exceptions                         | 14-98  |
|       | 14.14.2.5   | Description                        | 14-99  |
|       | 14.14.2.6   | Return Values                      | 14-99  |
| 14.15 | POAMana     | ager Member Functions              | 14-99  |
| 14    | 1.15.1 Port | ableServer::POAManager::activate   | 14-99  |
|       | 14.15.1.1   | Synopsis                           | 14-100 |
|       | 14.15.1.2   | C++ Binding                        | 14-100 |
|       | 14.15.1.3   | Argument                           | 14-100 |
|       | 14.15.1.4   | Exceptions                         | 14-100 |
|       | 14.15.1.5   | Description                        | 14-100 |
|       | 14.15.1.6   | Return Values                      | 14-100 |
| 14    | 1.15.2 Port | ableServer::POAManager::deactivate | 14-100 |
|       | 14.15.2.1   | Synopsis                           | 14-100 |
|       | 14.15.2.2   | C++ Binding                        | 14-101 |
|       | 14.15.2.3   | Argument                           | 14-101 |
|       | 14.15.2.4   | Exceptions                         | 14-101 |
|       | 14.15.2.5   | Description                        | 14-101 |
|       | 14.15.2.6   | Return Values                      | 14-101 |
| 14.16 | POA Polic   | cy Member Objects                  | 14-101 |
| 14    | I.16.1 Port | ableServer::LifespanPolicy         | 14-102 |
|       | 14.16.1.1   | Synopsis                           | 14-102 |
|       | 14.16.1.2   | Description                        | 14-102 |
|       | 14.16.1.3   | Exceptions                         | 14-102 |
| 14    | 1.16.2 Port | ableServer::IdAssignmentPolicy     | 14-102 |
|       | 14.16.2.1   | Synopsis                           | 14-102 |
|       | 14.16.2.2   | Description                        | 14-102 |
| 14.17 | Request M   | Member Functions                   | 14-103 |
| 14    | I.17.1 COF  | RBA::Request::arguments            | 14-104 |
|       | 14.17.1.1   | Synopsis                           | 14-104 |
|       | 14.17.1.2   | C++ Binding                        | 14-104 |
|       | 14.17.1.3   | Arguments                          | 14-104 |
|       | 14.17.1.4   | Description                        | 14-104 |
|       | 14.17.1.5   | Return Values                      | 14-104 |
| 14    | I.17.2 COF  | RBA::Request::ctx(Context_ptr)     | 14-105 |
|       | 14.17.2.1   | Synopsis                           | 14-105 |
|       | 14.17.2.2   | C++ Binding                        | 14-105 |
|       | 14.17.2.3   | Argument                           | 14-105 |
|       | 14.17.2.4   | Description                        | 14-105 |
|       | 14.17.2.5   | Return Values                      | 14-105 |
|       | 14.17.2.6   | See Also                           | 14-105 |
| 14    | I.17.3 COF  | RBA::Request::get_response         | 14-105 |



| 14.17.3.1   | Synopsis                    | 14-106 |
|-------------|-----------------------------|--------|
| 14.17.3.2   | C++ Binding                 | 14-106 |
| 14.17.3.3   | Arguments                   | 14-106 |
| 14.17.3.4   | Description                 | 14-106 |
| 14.17.3.5   | Return Values               | 14-106 |
| 14.17.3.6   | See Also                    | 14-106 |
| 14.17.4 COF | RBA::Request::invoke        | 14-106 |
| 14.17.4.1   | Synopsis                    | 14-106 |
| 14.17.4.2   | C++ Binding                 | 14-106 |
| 14.17.4.3   | Arguments                   | 14-106 |
| 14.17.4.4   | Description                 | 14-107 |
| 14.17.4.5   | Return Values               | 14-107 |
| 14.17.5 COF | RBA::Request::operation     | 14-107 |
| 14.17.5.1   | Synopsis                    | 14-107 |
| 14.17.5.2   | C++ Binding                 | 14-107 |
| 14.17.5.3   | Arguments                   | 14-107 |
| 14.17.5.4   | Description                 | 14-107 |
| 14.17.5.5   | Return Values               | 14-107 |
| 14.17.6 COF | RBA::Request::poll_response | 14-107 |
| 14.17.6.1   | Synopsis                    | 14-108 |
| 14.17.6.2   | C++ Binding                 | 14-108 |
| 14.17.6.3   | Arguments                   | 14-108 |
| 14.17.6.4   | Description                 | 14-108 |
| 14.17.6.5   | Return Values               | 14-108 |
| 14.17.6.6   | See Also                    | 14-108 |
| 14.17.7 COF | RBA::Request::result        | 14-108 |
| 14.17.7.1   | Synopsis                    | 14-108 |
| 14.17.7.2   | C++ Binding                 | 14-109 |
| 14.17.7.3   | Arguments                   | 14-109 |
| 14.17.7.4   | Description                 | 14-109 |
| 14.17.7.5   | Return Values               | 14-109 |
| 14.17.8 COF | RBA::Request::env           | 14-109 |
| 14.17.8.1   | Synopsis                    | 14-109 |
| 14.17.8.2   | C++ Binding                 | 14-109 |
| 14.17.8.3   | Arguments                   | 14-109 |
| 14.17.8.4   | Description                 | 14-109 |
| 14.17.8.5   | Return Values               | 14-109 |
| 14.17.9 COF | RBA::Request::ctx           | 14-110 |
| 14.17.9.1   | Synopsis                    | 14-110 |
| 14.17.9.2   | C++ Binding                 | 14-110 |
| 14.17.9.3   | Arguments                   | 14-110 |
| 14.17.9.4   | Description                 | 14-110 |



|       | 14.17.9.5  | Return Values            |     | 14-110 |
|-------|------------|--------------------------|-----|--------|
| 14.   | 17.10 CO   | RBA::Request::contexts   |     | 14-110 |
|       | 14.17.10.1 | Synopsis                 |     | 14-110 |
|       | 14.17.10.2 | C++ Binding              |     | 14-110 |
|       | 14.17.10.3 | Arguments                |     | 14-111 |
|       | 14.17.10.4 | Description              |     | 14-111 |
|       | 14.17.10.5 | Return Values            |     | 14-111 |
| 14.   | 17.11 COI  | RBA::Request::exceptions |     | 14-111 |
|       | 14.17.11.1 | Synopsis                 |     | 14-111 |
|       | 14.17.11.2 | C++ Binding              |     | 14-111 |
|       | 14.17.11.3 | Arguments                |     | 14-111 |
|       | 14.17.11.4 | Description              |     | 14-111 |
|       | 14.17.11.5 | Return Values            |     | 14-111 |
| 14.   | 17.12 CO   | RBA::Request::target     |     | 14-111 |
|       | 14.17.12.1 | Synopsis                 |     | 14-112 |
|       | 14.17.12.2 | C++ Binding              |     | 14-112 |
|       | 14.17.12.3 | Arguments                |     | 14-112 |
|       | 14.17.12.4 | Description              |     | 14-112 |
|       | 14.17.12.5 | Return Values            |     | 14-112 |
| 14.   | 17.13 CO   | RBA::Request::send_defer | red | 14-112 |
|       | 14.17.13.1 | Synopsis                 |     | 14-112 |
|       | 14.17.13.2 | C++ Binding              |     | 14-112 |
|       | 14.17.13.3 | Arguments                |     | 14-112 |
|       | 14.17.13.4 | Description              |     | 14-113 |
|       | 14.17.13.5 | Return Values            |     | 14-113 |
|       | 14.17.13.6 | See Also                 |     | 14-113 |
| 14.   | 17.14 CO   | RBA::Request::send_onew  | ay  | 14-113 |
|       | 14.17.14.1 | Synopsis                 |     | 14-113 |
|       | 14.17.14.2 | C++ Binding              |     | 14-113 |
|       | 14.17.14.3 | Arguments                |     | 14-113 |
|       | 14.17.14.4 | Description              |     | 14-113 |
|       | 14.17.14.5 | Return Values            |     | 14-113 |
|       | 14.17.14.6 | See Also                 |     | 14-114 |
| 14.18 | Strings    |                          |     | 14-114 |
| 14.   | 18.1 COR   | BA::string_alloc         |     | 14-114 |
|       | 14.18.1.1  | Synopsis                 |     | 14-115 |
|       | 14.18.1.2  | C++ Binding              |     | 14-115 |
|       | 14.18.1.3  | Argument                 |     | 14-115 |
|       | 14.18.1.4  | Description              |     | 14-115 |
|       | 14.18.1.5  | Return Values            |     | 14-115 |
|       | 14.18.1.6  | Example                  |     | 14-115 |
|       | 14.18.1.7  | See Also                 |     | 14-115 |



| 14    | .18.2 C   | ORBA::string_dup               | 14-115 |
|-------|-----------|--------------------------------|--------|
|       | 14.18.2.  | 1 Synopsis                     | 14-116 |
|       | 14.18.2.  | 2 C++ Binding                  | 14-116 |
|       | 14.18.2.  | 3 Argument                     | 14-116 |
|       | 14.18.2.  | 4 Description                  | 14-116 |
|       | 14.18.2.  | 5 Return Values                | 14-116 |
|       | 14.18.2.  | 6 Example                      | 14-116 |
|       | 14.18.2.  | 7 See Also                     | 14-116 |
| 14    | .18.3 C   | ORBA::string_free              | 14-116 |
|       | 14.18.3.  | 1 Synopsis                     | 14-117 |
|       | 14.18.3.  | 2 C++ Binding                  | 14-117 |
|       | 14.18.3.  | 3 Argument                     | 14-117 |
|       | 14.18.3.  | 4 Description                  | 14-117 |
|       | 14.18.3.  | 5 Return Values                | 14-117 |
|       | 14.18.3.  | 6 Example                      | 14-117 |
|       | 14.18.3.  | 7 See Also                     | 14-117 |
| 14.19 | Wide St   | rings                          | 14-117 |
| 14.20 | TypeCo    | de Member Functions            | 14-118 |
| 14    | .20.1 M   | emory Management               | 14-119 |
|       | 14.20.1.  | 1 CORBA::TypeCode::equal       | 14-119 |
|       | 14.20.1.  | 2 CORBA::TypeCode::id          | 14-120 |
|       | 14.20.1.  | 3 CORBA::TypeCode::kind        | 14-120 |
|       | 14.20.1.  | 4 CORBA::TypeCode::param_count | 14-122 |
|       | 14.20.1.  | CORBA::TypeCode::parameter     | 14-122 |
| 14.21 | Exception | on Member Functions            | 14-123 |
| 14.22 | Standar   | d Exceptions                   | 14-125 |
| 14    | .22.1 E   | cception Definitions           | 14-125 |
| 14    | .22.2 O   | oject Nonexistence             | 14-126 |
| 14    | .22.3 Tr  | ansaction Exceptions           | 14-126 |
| 14.23 | Exception | onList Member Functions        | 14-127 |
| 14    | .23.1 C   | ORBA::ExceptionList::count     | 14-127 |
|       | 14.23.1.  | 1 Synopsis                     | 14-127 |
|       | 14.23.1.  | 2 C++ Binding                  | 14-127 |
|       | 14.23.1.  | 3 Arguments                    | 14-128 |
|       | 14.23.1.  | 4 Exception                    | 14-128 |
|       | 14.23.1.  | 5 Description                  | 14-128 |
|       | 14.23.1.  | 6 Return Values                | 14-128 |
| 14    | .23.2 C   | ORBA::ExceptionList::add       | 14-128 |
|       | 14.23.2.  | 1 Synopsis                     | 14-128 |
|       | 14.23.2.  | 2 C++ Binding                  | 14-128 |
|       | 14.23.2.  | 3 Arguments                    | 14-128 |
|       | 14.23.2.  | 4 Exception                    | 14-128 |
|       |           |                                |        |



|    | 14.23.2.5  | Description                     | 14-128 |
|----|--|---------------------------------|--------|
|    | 14.23.2.6  | Return Values                   | 14-129 |
|    | 14.23.2.7  | See Also                        | 14-129 |
|    | 14.23.3 CO   | RBA::ExceptionList::add_consume | 14-129 |
|    | 14.23.3.1  | Synopsis                        | 14-129 |
|    | 14.23.3.2  | C++ Binding                     | 14-129 |
|    | 14.23.3.3  | Arguments                       | 14-129 |
|    | 14.23.3.4  | Exceptions                      | 14-129 |
|    | 14.23.3.5  | Description                     | 14-129 |
|    | 14.23.3.6  | Return Values                   | 14-129 |
|    | 14.23.3.7  | See Also                        | 14-130 |
|    | 14.23.4 CO   | RBA::ExceptionList::item        | 14-130 |
|    | 14.23.4.1  | Synopsis                        | 14-130 |
|    | 14.23.4.2  | C++ Binding                     | 14-130 |
|    | 14.23.4.3  | Argument                        | 14-130 |
|    | 14.23.4.4  | Exceptions                      | 14-130 |
|    | 14.23.4.5  | Description                     | 14-130 |
|    | 14.23.4.6  | Return Values                   | 14-130 |
|    | 14.23.4.7  | See Also                        | 14-130 |
|    | 14.23.5 <en< td=""><td>iter Topic Title Here&gt;</td><td>14-131</td></en<> | iter Topic Title Here>          | 14-131 |
|    | 14.23.5.1  | Synopsis                        | 14-131 |
|    | 14.23.5.2  | C++ Binding                     | 14-131 |
|    | 14.23.5.3  | Argument                        | 14-131 |
|    | 14.23.5.4  | Exceptions                      | 14-131 |
|    | 14.23.5.5  | Description                     | 14-131 |
|    | 14.23.5.6  | Return Values                   | 14-131 |
|    | 14.23.5.7  | See Also                        | 14-131 |
| 15 | Server Side N  | Mapping                         |        |
|    | 15.1 Implementi  | ng Interfaces                   | 15-1   |
|    | 15.2 Inheritance   | -based Interface Implementation | 15-1   |
|    | 15.3 Delegation-   | -based Interface Implementation | 15-3   |
|    | 15.4 Implementi  | ng Operations                   | 15-6   |



#### List of Figures

| 5-1 | Registering a Factory Object           | 5-3 |
|-----|--|-----|
| 5-2 | Locating a Factory Object              | 5-4 |
| 5-3 | Inter-Domain FactoryFinder Interaction | 5-7 |



#### List of Tables

| 1-1   | Predefined Macros  | 1-2    |
|-------|--|--------|
| 1-2   | C++ IDL Compiler   | 1-2    |
| 4-1   | Oracle Remote Clients Supported  | 4-6    |
| 4-2   | Returned Object References   | 4-7    |
| 4-3   | Tobj Module Exceptions   | 4-7    |
| 4-4   | Differences Between corbaloc and corbalocs URL Address Formats           | 4-12   |
| 4-5   | Initialize Exceptions  | 4-19   |
| 4-6   | CreateObject Exceptions  | 4-20   |
| 4-7   | Returned Object References   | 4-23   |
| 4-8   | BNF Format for URL Elements  | 4-25   |
| 4-9   | Descriptions of URL Elements   | 4-26   |
| 4-10  | BNF Syntax for URL   | 4-28   |
| 4-11  | Supported INS Interfaces and Operations                                  | 4-31   |
| 13-1  | Basic OMG IDL and C++ Data Types   | 13-3   |
| 13-2  | Object, Pseudo-object, and User-defined OMG IDL and C++ Types            | 13-3   |
| 13-3  | Comparison of Operators Supported for User-defined Data Type var Classes | 13-44  |
| 13-4  | Operator Signatures for _var Classes                                     | 13-44  |
| 13-5  | Comparison of Operators Supported for User-defined Data Type Out Classes | 13-45  |
| 13-6  | Operator Signatures for _out Classes                                     | 13-46  |
| 13-7  | Basic Argument and Result Passing  | 13-53  |
| 13-8  | T_var Argument and Result Passing  | 13-54  |
| 13-9  | Caller Argument Storage Responsibilities                                 | 13-54  |
| 13-10 | Argument Passing Cases   | 13-55  |
| 14-1  | Exception and Reasons  | 14-56  |
| 14-2  | Minor Codes for CORBA::BAD_PARAM Standard Exception                      | 14-76  |
| 14-3  | Legal Typecode Kinds and Parameters                                      | 14-121 |
| 14-4  | Exception Definitions  | 14-125 |



### **Preface**

The Oracle Tuxedo CORBA Programming Reference guide provides detailed information about various CORBA Programming references.



# **Documentation Accessibility**

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

#### **Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <a href="http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info</a> or visit <a href="http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs</a> if you are hearing impaired.



1

## OMG IDL Syntax and the C++ IDL Compiler

The Object Management Group (OMG) Interface Definition Language (IDL) is used to describe the interfaces that client objects call and that object implementations provide. An OMG IDL interface definition fully specifies each operation's parameters and provides the information needed to develop client applications that use the interface's operations.

Client applications are written in languages for which mappings from OMG IDL statements have been defined. How an OMG IDL statement is mapped to a client language construct depends on the facilities available in the client language. For example, an OMG IDL exception might be mapped to a structure in a language that has no notion of exception, or to an exception in a language that does.

OMG IDL statements obey the same lexical rules as C++ statements, although new keywords are introduced to support distribution concepts. OMG IDL statements also provide full support for standard C++ preprocessing features and OMG IDL-specific pragmas.



When using a pragma version statement, be sure to locate it after the corresponding interface definition. The following is an example of proper usage:

```
module A
{
interface B
{
#pragma version B "3.5"
void op1();
};
};
```

The OMG IDL grammar is a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. OMG IDL is a declarative language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables.

For a description of OMG IDL grammar, see Chapter 3 of the Common Object Request Broker: Architecture and Specification Revision 2.4 "OMG IDL Syntax and Semantics."

All OMG IDL grammar is supported, with the exception of the following type declarations and associated literals:

native

Note:

Because CORBA 2.4 states that the native type declaration is intended for use in Object Adapters, not user interfaces, this type is available in the PortableServer module only for clients that support callbacks, that is, joint client/servers.

- long double
- fixed

Do not use these data types in IDL definitions.



Support for the long long, unsigned long long, wchar, and wstring data types was added to Oracle Tuxedo CORBA in release 8.0.

- OMG IDL Compiler Extensions
- C++ IDL Compiler Constraints

#### 1.1 OMG IDL Compiler Extensions

The IDL compiler defines preprocessor macros specific to the platform. All macros predefined by the preprocessor that you are using can be used in the OMG IDL file, in addition to the user-defined macros. You can also define your own macros when you are compiling or loading OMG IDL files.

The following table describes the predefined macros for each platform.

Table 1-1 Predefined Macros

| Macro Identifier | Platform on Which the Marco Is Defined |
|------------------|--|
| unix             | Sun Solaris, HP-UX, and IBM AIX        |
| sun              | Sun Solaris                            |
| hpux             | HP-UX                                  |
| aix              | IBM AIX                                |
| win_nt           | Microsoft Windows                      |

#### 1.2 C++ IDL Compiler Constraints

In the following table describes constraints for the Oracle Tuxedo 9.1 C++ IDL compiler and provides information about recommended workarounds.

Table 1-2 C++ IDL Compiler

### Table 1-2 (Cont.) C++ IDL Compiler

#### Description

A warning is generated by the C++ IDL compiler when context strings that contain wildcard characters are used in the operation definitions. When you specify a context string in an OMG IDL operation definition, the following warning may be generated:

#### Workaround

The OMG CORBA specification is ambiguous about whether the first character of a context string must be alphabetic.

This warning is generated to inform you that you are not in compliance with some interpretations of the OMG CORBA specification. If you are intending to specify all strings as context string values, as shown above, the OMG CORBA specification requires a comma-separated list of strings, in which the first character is alphabetic.



The example shown above is not OMG CORBA compliant, but it is processed by the Oracle Tuxedo software as intended by the user.

#### Constraint

#### Use of wildcarding in OMG IDL context strings produces warnings.

### Description

A warning is generated by the C++ IDL compiler when context strings that contain wildcard characters are used in the operation definitions. When you specify a context string in an OMG IDL operation definition, the following warning may be generated:



#### Table 1-2 (Cont.) C++ IDL Compiler

#### Workround

The OMG CORBA specification is ambiguous about whether the first character of a context string must be alphabetic.

This warning is generated to inform you that you are not in compliance with some interpretations of the OMG CORBA specification. If you are intending to specify all strings as context string values, as shown above, the OMG CORBA specification requires a comma-separated list of strings, in which the first character is alphabetic.



The example shown above is not OMG CORBA compliant, but it is processed by the Oracle Tuxedo software as intended by the user.

#### Constraint

The C++ IDL compiler does not support some data types.

Description

The C++ IDL compiler currently does not support the following data types, which are defined in the CORBA specification version 2.4:

- native
- fixed
- long double

**Workaround** Avoid using these data types in IDL definitions.

#### Constraint

Using certain substrings in identifiers may cause incorrect code generation by the C++ IDL compiler.

Description

Using the following substrings in identifiers may cause code to be generated incorrectly and result in errors when the generated code is compiled:

```
get
set
Impl
ptr
slice
```

Workround

Avoid the use of these substrings in identifiers.

### Constraint

Inconsistent behavior in IDL compiler regarding case sensitivity.

Description

According to the CORBA standard, IDL identifiers that differ only in case should be considered colliding and yield a compilation error. There is a current limitation of the Oracle Tuxedo IDL compiler for C++ bindings in that it does not always detect and report such name collisions except for value type. Value type will follow CORBA standard regarding case sensitivity.

**Workaround** Avoid using IDL identifiers that differ only in case.

#### Constraint

C++ IDL typedef problem.

### Table 1-2 (Cont.) C++ IDL Compiler

#### Description

The C++ IDL compiler generates code that does not compile when:

- Defining IDL variables of char or boolean type
- And the type is aliased multiple times

For example, the generated C++ code from the following IDL code will not compile:

```
module X {
         typedef boolean a;
         typedef a b;
         interface Y {
               attribute b Z;
         };
};
```

C++ compilers report an error that an "operator<<" is ambiguous and that there is no "operator>>" for type char. These errors are produced because of the multiple levels of typedefs; the C++ compiler may not associate the type X::b with CORBA::Boolean because of the intermediate type definition of X::a.

#### Workaround

Use a single level of indirection when you define <code>char</code> or <code>boolean</code> types. In the IDL example above, the attribute 'X::Z' would be defined using either the standard type 'boolean' or the user type 'X::a', but not the user type 'X::b'.



# Implementation Configuration File (ICF)

The Oracle Tuxedo CORBA TP Framework application programming interface (API) provides callback methods for object activation and deactivation. These methods provide the ability for application code to implement flexible state management schemes for CORBA objects.

State management is the way you control the saving and restoring of object state during object deactivation and activation. State management also affects the duration of object activation, which influences the performance of servers and their resource usage. The external API of the TP Framework includes the activate\_object() and deactivate\_object() methods, which provide a possible location for state management code. Additionally, the TP Framework API includes the deactivateEnable() method to enable the user to control the timing of object deactivation. The default duration of object activation is controlled by policies assigned to implementations at OMG IDL compile time.

While CORBA objects are active, their state is contained in a servant. This state must be initialized when objects are first invoked (that is, the first time a method is invoked on a CORBA object after its object reference is created) and on subsequent invocations after objects have been deactivated.

While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. When an object is activated, its state must be restored. The object's state can be saved in shared memory, in a file, in a database, and so forth. It is up to the programmer to determine what constitutes an object's state, and what must be saved before an object is deactivated and restored when an object is activated.

You can use the Implementation Configuration File (ICF) to set activation policies to control the duration of object activations in each implementation. The ICF file manages object state by specifying the activation policy. The activation policy determines the in-memory activation duration for a CORBA object. A CORBA object is active in a Portable Object Adapter (POA) if the POA's active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of an object ID with its active servant.

- ICF Syntax
- Sample ICF File
- · Creating the ICF File

# 2.1 ICF Syntax

ICF syntax is as follows:

```
[activation_policy(method | transaction | process);]
[transaction_policy(never | ignore | optional | always);]
[concurrency_policy(user_controlled | system_controlled);]
    [retry_policy(never | always)];
};[];]
```

#### pragmas

The four optional pragmas allow you to set a specific policy as the default policy for the entire ICF for all implementations that do not have an explicit activation policy,

transaction\_policy, concurrency\_policy, or retry\_policy statement. This feature relieves the programmer from having to specify policies for each implementation and/or allows overriding of the defaults.

#### Module module-name

The module-name variable is optional if it is optional in the OMG IDL file. This variable is used for scoping and grouping. Its use must be consistent with the way it is used inside the OMG IDL file.

#### implementation-name

This variable is optional and is used as the name of the servant or as the class name in the server. It is constructed using interface-name with an \_i appended if it is not specified by the programmer.

#### implements (module-name::interface-name)

This variable identifies the module and the interface to which the activation policy and the transaction policy apply.

### activation\_policy

For a description of the activation policies, see Activation Policy.

#### transaction policy

For a description of the transaction policies, see Transaction Policies.

#### concurrency\_policy

For description of the concurrency policies, see Parallel Objects.

#### retry policy

For a description of the retry policy, see IIOP Client Failover.

# 2.2 Sample ICF File

The following code snippet shows a sample ICF file.

```
module POA_University1 {
    implementation CourseSynopsisEnumerator_i {
        activation_policy(process);
        transaction_policy(optional);
        implements(University1::CourseSynopsisEnumerator);
    };
};
module POA_University1 {
    implementation Registrar_i {
        activation_policy(method);
        transaction_policy(optional);
        implements(University1::Registrar);
    };
```

```
};
module POA_University1 {
    implementation RegistrarFactory_i {
        activation_policy(process);
        transaction_policy(optional);
        implements(University1::RegistrarFactory);
    };
};
```

# 2.3 Creating the ICF File

You have the option of either coding the ICF file manually or using the <code>genicf</code> command to generate it from the OMG IDL file. For a description of the syntax and options for the <code>genicf</code> command, see the Oracle Tuxedo Command Reference.

✓ See Also:

Steps for Creating a Oracle Tuxedo CORBA Server Application in Creating CORBA Server Applications.



# TP Framwork

This topic includes the following sections:

- A Simple Programming Model. This section describes:
  - Control Flow
  - Object State Management
  - Transaction Integration
  - Object Housekeeping
  - High-level Services
- · State Management. This section describes:
  - Activation Policy
  - Application-controlled Activation and Deactivation
  - Servant Lifetime
  - Saving and Restoring Object State
- Transactions. This section describes:
  - Transaction Policies
  - Transaction Initiation
  - Transaction Termination
  - Transaction Suspend and Resume
  - Restrictions on Transactions
  - SQL and Global Transactions
  - Voting on Transaction Outcome
  - Transaction Timeouts
- IIOP Client Failover
- WebLogic CORBA Clustering and Load Balancing Support
- Parallel Objects
- TP Framework API
- · Error Conditions, Exceptions, and Error Messages

The Oracle Tuxedo CORBA TP Framework provides a programming TP Framework that enables users to create servers for high-performance TP applications. This chapter describes the TP Framework programming model and the TP Framework application programming interface (API) in detail. Additional information about how to use this API can be found in *Creating CORBA Server Applications*.

The TP Framework is required when developing Oracle Tuxedo CORBA servers. Later releases will relax this requirement, though it is expected that most customers will use the TP Framework as an integral part of their applications.

Oracle Tuxedo provides the infrastructure for providing load balancing, transactional capabilities, and administrative infrastructure. The base API used by the TP Framework is the CORBA API with Oracle extensions. The TP Framework API is exposed to customers. The Oracle Tuxedo ATMI is an optional API that can be mixed in with TP Framework APIs, allowing a customer to deploy distributed applications using a mix of CORBA servers and ATMI servers.

Before Oracle Tuxedo CORBA, ORB products did not approach Oracle Tuxedo's performance in large-scale environments. Oracle Tuxedo systems support applications that can process hundreds of transactions per second. These applications are built using the Oracle Tuxedo stateless-service programming model that minimizes the amount of system resources used for each request, and thus maximizes throughput and price performance.

Now, Oracle Tuxedo CORBA and its TP Framework give customers a way to develop CORBA applications with performance similar to Oracle Tuxedo ATMI applications. Oracle Tuxedo CORBA servers provide throughput, response time, and price performance approaching the Oracle Tuxedo stateless-service programming model, while using the CORBA programming model.

- A Simple Programming Model
- State Management
- Transactions
- IIOP Client Failover
- WebLogic CORBA Clustering and Load Balancing Support
- Parallel Objects
- TP Framework API
- CosTransactions::TransactionalObject Interface Not Enforced
- Error Conditions, Exceptions, and Error Messages

# 3.1 A Simple Programming Model

The TP Framework provides a simple, useful subset of the wide range of possible CORBA object implementation choices. You use it for the development of server-side object implementations only. When using any client-side CORBA ORB, clients interact with CORBA objects whose server-side implementations are managed by the TP Framework. Clients are unaware of the existence of the TP Framework—a client written to access a CORBA object executing in a non-Oracle Tuxedo server environment will be able to access that same CORBA object executing in an Oracle Tuxedo server environment without any changes or restrictions to the client interface.

The TP Framework provides a server environment and an API that is easier to use and understand than the CORBA Portable Object Adapter (POA) API, and is specifically geared towards enterprise applications. It is a simple server programming model and an orthodox implementation of the CORBA model, which will be familiar to programmers using ORBs such as ORBIX or VisiBroker.

The TP Framework simplifies the programming of Oracle Tuxedo CORBA servers by reducing the complexity of the server environment in the following ways:

- The TP Framework does all interactions with the POA and the Naming Service. The application programmer requires no knowledge of POA or Naming Service interfaces.
- The TP Framework is single threaded—only one request on one CORBA object will be processed at a time, obviating the need to write thread-safe implementations.



 A CORBA object may be involved in only one transaction at a time (consistent with the association of one object ID to one servant).

The TP Framework provides the following functionality:

- Control Flow
- Object State Management
- Transaction Integration
- Object Housekeeping
- High-level Services

# 3.1.1 Control Flow

The TP Framework, in conjunction with the ORB and the POA, controls the flow of the application program by doing the following:

- Controlling the server mainline and invoking callback methods on TP Framework-defined classes at appropriate times for server startup and shutdown. This relieves the application programmer from complex interactions related to ORB and POA initialization and coordination of transactions, resource managers, and object state on shutdown.
- Scheduling objects for activation and deactivation when client requests arrive and are completed. This removes the complexity of management of object activation and deactivation from the realm of the application programmer and enables the use of the TP monitor infrastructure's powerful load-balancing capabilities, crucial to performance of mission-critical tasks.

# 3.1.2 Object State Management

The TP Framework API provides callback methods for application code to implement flexible state management schemes for CORBA objects. State management involves the saving and restoring of object state on object deactivation and activation. It also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The default duration of object activation is controlled by policies assigned to implementations at IDL compile time.

# 3.1.3 Transaction Integration

TP Framework transaction integration provides the following features:

- CORBA objects can participate in global transactions.
- Objects participating in transactions can be implemented as stateful objects that remain in memory for the duration of a transaction (by using the transaction activation policy), to decrease client response time.
- CORBA objects that participate in transactions can affect transaction outcome either during their transactional work or just prior to the system's execution of the two-phase commit algorithm after all transactional work has been completed.
- Transactions can be automatically initiated on the server transparent to the client.

# 3.1.4 Object Housekeeping

When a server is shut down, the TP Framework rolls back any transactions that the server is involved in and deactivates any CORBA objects that are currently active.

# 3.1.5 High-level Services

The TP interface in the TP Framework API provides methods for performing object registrations and utility functions. The following services are provided:

- Object reference creation
- · Factory-based routing support
- Accessors for system objects, such as the ORB
- Registration and unregistration of factories with the FactoryFinder
- Application-controlled activation and deactivation
- User logging

The purpose of these high-level service methods is to eliminate the need for developers to understand the CORBA POA, CORBA Naming Service, and Oracle Tuxedo APIs, which they use for their underlying implementations. By encapsulating the underlying API calls with a high-level set of methods, programmers can focus their efforts on providing business logic rather than understanding and using the more complex underlying facilities.

# 3.2 State Management

State management involves the saving and restoring of object state on object deactivation and activation. It also concerns the duration of activation of objects, which influences the performance of servers and their resource usage. The external API of the TP Framework provides activate\_object and deactivate\_object methods, which are a possible location for state management code.

- Activation Policy
- Application-controlled Activation and Deactivation
- Servant Lifetime
- Saving and Restoring Object State

# 3.2.1 Activation Policy

State management is provided in the TP Framework by the activation policy. This policy controls the activation and deactivation of servants for a particular IDL interface (as opposed to the creation and destruction of the servants). This policy is applicable only to CORBA objects using the TP Framework.

The activation policy determines the default in-memory activation duration for a CORBA object. A CORBA object is active in a POA if the POA's active object map contains an entry that associates an object ID with an existing servant. Object deactivation removes the association of an object ID with its active servant. You can choose from one of three activation policies:method (the default), transaction, or process.



The activation policies are set in an ICF file that is configured at OMG IDL compile time. For a description of the ICF file, refer to the Implementation Configuration File (ICF) section.



The activation policies are described below:

method (This is the default activation policy.)
 The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the method. At the completion of a method, the object is deactivated. When the next method is invoked on the object reference, the CORBA object is activated (the object ID is associated with a new servant). This behavior is similar to that of an Oracle Tuxedo stateless service.

#### transaction

The activation of the CORBA object (that is, the association between the object ID and the servant) lasts until the end of the transaction. During the transaction, multiple object methods can be invoked. The object is activated before the first method invocation on the object and is deactivated in one of the following ways:

- If a user-initiated transaction is in effect when the object is activated, the object is deactivated when the first of the following occurs: the transaction is committed or rolled back, or the server is shut down in an orderly fashion. The latter is done using either the tmshutdown or tmadmin command. These commands are described in the Oracle Tuxedo Command Reference online document.
- If a user-initiated transaction is not in effect when the TP object is activated, the TP object is deactivated when the method completes.

The transaction activation policy provides a means for an object to vote on the outcome of the transaction prior to the execution of the two-phase commit algorithm. An object votes to roll back the transaction by calling  $Current.rollback\_only()$  in the  $Tobj\_ServantBase::deactivate\_object$  method. It votes to commit the transaction by not calling  $Current.rollback\_only()$  in the method.

### Note:

This is a model of resource allocation that is similar to that of an Oracle Tuxedo conversational service. However, this model is less expensive than the Oracle Tuxedo conversational service in that it uses fewer system resources. This is because of the Oracle Tuxedo ORB's multicontexted dispatching model (that is, the presence of many servants in memory at the same time for one server), which makes it possible for a single server process to be shared by many concurrently active servants that service many clients. In the Oracle Tuxedo system, the process would be dedicated to a single client and to only one service for the duration of a conversation.

#### process

The activation of the CORBA object begins when it is invoked while in an inactive state and, by default, lasts until the end of the process.

#### Note:

The TP Framework API provides an interface method (TP::deactivateEnable) that allows the application to control the timing of object deactivation for objects that have the activation policy set to process. For a description of this method, see the section TP::deactivateEnable().



# 3.2.2 Application-controlled Activation and Deactivation

Ordinarily, activation and deactivation decisions are made by the TP Framework, as discussed earlier in this chapter. The techniques in this section show how to use alternate mechanisms. The application can control the timing of activation and deactivation explicitly for objects with particular policies.

- Explicit Activation
- Self Deactivation

# 3.2.2.1 Explicit Activation

Application code can bypass the on-demand activation feature of the TP Framework for objects that use the process activation policy. The application can "preactivate" an object (that is, activate it before any invocation) using the TP::create active object reference call.

Preactivation works as follows. Before the application creates an object reference, the application instantiates a servant and initializes that servant's state. The application uses <code>TP::create\_active\_object\_reference</code> to put the object into the Active Object Map (that is, associate the servant with an <code>ObjectId</code>). Then, when the first invocation is made, the TP Framework immediately directs the request to the process that created the object reference and then to the existing servant, bypassing the necessity to call <code>Server::create\_servant</code> and then the servant's <code>activate\_object</code> method (just as if this were the second or later invocation on the object). Note that the object reference for such an object will not be directed to another server and the object will never go through on-demand activation as long as the object remains activated.

Since the preactivated object has the process activation policy, it will remain active until one of two events occurs: (1) the ending of the process or (2) a TP::deactivateEnable call.

- Usage Notes
- Caution to Users

# 3.2.2.1.1 Usage Notes

Preactivation is especially useful if the application needs to establish the servant with an initial state in the same process, perhaps using shared memory to initialize state. Waiting to initialize state until a later time and in a potentially different process may be very difficult if that state includes pointers, object references, or complex data structures.

TP::create\_active\_object\_reference guarantees that the preactivated object is in the same process as the code that is doing the preactivation. While this is convenient, preactivation should be used sparingly, as should all process objects, because it preallocates precious resources. However, when needed and used properly, preallocation is more efficient than alternatives.

Examples of such usage might be an object using the "iterator" pattern. For example, there might a potentially long list of items that could be returned (in an unbound IDL sequence) from a "database\_query" method (for example, the contents of the telephone book). Returning all such items in the sequence is impractical because the message size and the memory requirements would be too large.

On an initial call to get the list, an object using the iterator pattern returns only a limited number of items in the sequence and also returns a reference to an "iterator" object that can be invoked to receive further elements. This iterator object is initialized by the initial object; that is, the initial object creates a servant and sets its state to keep track of where in the long list of



items the iteration currently stands (the pointer to the database, the query parameters, the cursor, and so forth).

The initial object preactivates this iterator object by using

TP::create\_active\_object\_reference. It also creates an object reference to that object to return to the client. The client then invokes repeatedly on the iterator object to receive, say, the next 100 items in the list each time. The advantage of preactivation in this situation is that the state might be complex. It is often easiest to set such state initially, from a method that has all the information in its context (call frame), when the initial object still has control.

When the client is finished with the iterator object, it invokes a final method on the initial object which deacativates the iterator object. The initial object deactivates the iterator object by invoking a method on the iterator object that calls the TP::deactivateEnable method, that is, the iterator object calls TP::deactivateEnable on itself.

### 3.2.2.1.2 Caution to Users

For objects to be preactivated in this fashion, the state usually cannot be recovered if a crash occurs. (This is because the state was considered too complex or inconvenient to set upon initial, delayed activation.) This is a valid object technique, essentially stating that the object is valid only for a single activation period.

However, a problem may arise because of the "one-time" usage. Since a client still holds an object reference that leads to the process containing that state, and since the state cannot be recreated after the crash, care must be taken that the client's next invocation does not automatically provoke a new activation of the object, because that object would have inapplicable state.

The solution is to refuse to allow the object to be activated automatically by the TP Framework. If the user provides the <code>TobjS::ActivateObjectFailed</code> exception to the TP Framework as a result of the <code>activate\_object</code> call, the TP Framework will not complete the activation and will return an exception to the client, <code>CORBA::OBJECT\_NOT\_EXIST</code>. The client has presumably been warned about this possibility, since it knows about the iterator (or similar) pattern. The client must be prepared to restart the iteration.

### Note:

This defensive measure may not be necessary in the future; the TP Framework itself may detect that the object reference is no longer valid. In particular, you should not depend on the possibility that the activate\_object method might be called. If the TP Framework does in fact change, activate\_object will not be called and the framework itself will generate the <code>OBJECT\_NOT\_EXIST</code> exception.

### 3.2.2.2 Self Deactivation

Just as it is possible to preactivate an object with the process activation policy, it is possible to request the deactivation of an object with the process activation policy. The ability to preactivate and the ability to request deactivation are independent; regardless of how an object was activated, it can be deactivated explicitly.

A method in the application can request (via TP::deactivateEnable) that the object be deactivated. When TP::deactivateEnable is called and the object is subsequently deactivated, no guarantee is made that subsequent invocations on the CORBA object will result in reactivation in the same process as a previous activation. The association between the ObjectId and the servant exists from the activation of the CORBA object until one of the



following events occurs: (1) the shutdown of the server process or (2) the application calls <code>TP::deactivateEnable</code>. After the association is broken, when the object is invoked again, it can be reactivated anywhere that is allowed by the Oracle Tuxedo configuration parameters.

There are two forms of TP::deactivateEnable. In the first form (with no parameters), the object currently executing will be deactivated after completion of the method in which the call is made. The object itself makes the decision that it should be deactivated. This is often done during a method call that acts as a "signoff" signal.

The second form of TP::deactivateEnable allows a server to request deactivation of any active object, whether it is the object that is executing or not; that is, any part of the server can ask that the object be deactivated. This form takes parameters identifying the object to be deactivated. Explicit deactivation is not allowed for objects with an activation policy of transaction, because such objects cannot be safely deactivated until the end of a transaction.

In the TP::deactivateEnable call, the TP Framework calls the servant's deactivate\_object method. Exactly when the TP Framework invokes deactivate\_object depends on the state of the object to be deactivated. If the object is not currently in execution, the TP Framework deactivates it before returning to the caller. The object might be currently executing a method; this is always the case for TP::deactivateEnable with no parameters (since it refers to the currently executing object). In this case, TP::deactivateEnable is not told whether the object was deactivated immediately or not.

### Note:

The TP::deactivateEnable(interface, object id, servant) method can be used to deactivate an object. However, if that object is currently in a transaction, the object will be deactivated when the transaction commits or rolls back. If an invoke occurs on the object before the transaction is committed or rolled back, the object will not be deactivated. To ensure the desired behavior, make sure that the object is not in a transaction or ensure that no invokes occur on the object after the TP::deactivateEnable() call until the transaction is complete.

# 3.2.3 Servant Lifetime

A servant is a C++ class that contains methods to implement an IDL interface's operations. The user writes the servant code. The TP Framework invokes methods in the servant code to satisfy requests. The servant is created by the C++ "new" statement and is destroyed by the C++ "delete" statement. Exactly who does the creation and who does the deletion, and the timing of creation and deletion, is the subject of this section.

- The Normal Case
- Special Cases

### 3.2.3.1 The Normal Case

In the normal case, the TP Framework completely controls the lifetime of a servant. The basic model is that, when a request for an inactive object arrives, the TP Framework obtains a servant and then activates it (by calling its activate\_object method). At deactivation time, the TP Framework calls the servant's deactivate\_object method and then disposes of the servant.

The phase "the TP Framework obtains a servant" means that when the TP Framework needs a servant to be created, it calls a user-written Server method, either Server::create\_servant or

ServerBase::create\_servant\_with\_id. At that time, the application code must return a pointer to the requested servant. The application almost always does this by using the C++ "new" statement to create a new instance of a servant. The phrase "disposes of the servant" means that the TP Framework removes the reference to the servant, which actually deletes it.

The application must be aware that this current behavior of always creating and removing a servant may change in future versions of this product. The application should not depend on the current behavior, but should write servant code that allows reuse of a servant. Specifically, the servant code must work even if the servant has not been freshly created (by the C++ "new" statement). The TP Framework reserves the right not to remove a servant after it has been deactivated and then to reactivate it. This means that the servant must completely initialize itself at the time of the callback on the servant's activate\_object method, not at the time of servant creation (not in the constructor).

# 3.2.3.2 Special Cases

There are two techniques an application can use to alter the normal TP Framework use of servants. The first has to do with obtaining a servant and the second has to do with disposing of the servant.

The application can alter the "obtaining" mechanism by using explicit preactivation. In this case, the application creates and initializes a servant before asking the TP Framework to declare it activated. Once such a servant has been turned over to the TP Framework (by the TP::create\_active\_object\_reference call), that servant is treated by the TP Framework just like every other servant. The only difference is in its method of creation and initialization.

The application can alter the "disposing" mechanism by taking the responsibility for disposing of a servant instead of leaving that responsibility with the TP Framework. Once a servant is known to the TP Framework (through Server::create servant,

ServerBase::create\_servant\_with\_id, or TP::create\_active\_object\_reference), the TP Framework's default behavior is to remove that servant itself. In this case, the application code must no longer use references to the servant after deactivation.

However, the application may tell the TP Framework not to dispose of the servant after the TP Framework deactivates it. Taking responsibility for a servant is done on an individual servant basis, not for a whole class of servants, by calling <code>Tobj\_ServantBase::\_add\_ref</code> with a parameter identifying the servant.

#### Note:

In applications written using Oracle Tuxedo release 8.0 or later, use the <code>Tobj\_ServantBase::\_add\_ref</code> method instead of the <code>TP::application\_responsibility()</code> method. Unlike the <code>TP::application\_responsibility()</code> method, the <code>add\_ref()</code> method takes no arguments.

The advantage of the application taking responsibility for the servant is that the servant does not have to be created anew. If obtaining the servant is an expensive proposition, the application may choose to save the servant and reuse it later. This is especially likely to be true for servants for preactivated objects, but is true in general. For example, the next time the TP Framework makes a call on Server::create servant

orServerBase::create\_servant\_with\_id, the application might return a previously saved servant.



Additionally, once an application has taken responsibility for a servant, the application must take care to remove the servant (using Tobj\_ServantBase::\_remove\_ref) when the servant is no longer needed, that is, when the reference count drops to zero, the same as for any other C++ instance. For more information about how the \_remove\_ref() method works, see Tobj\_ServantBase::\_remove\_ref().

For more information on writing single-threaded and multithreaded server applications, see *Creating CORBA Server Applications*.

# 3.2.4 Saving and Restoring Object State

While CORBA objects are active, their state is contained in a servant. Unless an application uses TP::create\_active\_object\_reference, state must be initialized when the object is first invoked (that is, the first time a method is invoked on a CORBA object after its object reference is created), and on subsequent invocations after they have been deactivated. While a CORBA object is deactivated, its state must be saved outside the process in which the servant was active. The object's state can be saved in shared memory, in a file, or in a database. Before a CORBA object is deactivated, its state must be saved, and when it is activated, its state must be restored.

The programmer determines what constitutes an object's state and what must be saved before an object is deactivated, and restored when an object is activated.

Note On Use of Constructors and Destructors for CORBA Objects

# 3.2.4.1 Note On Use of Constructors and Destructors for CORBA Objects

The state of CORBA objects must not be initialized, saved, or restored in the constructors or destructors for the servant classes. This is because the TP Framework may reuse an instance of a servant rather than deleting it at deactivation. No guarantee is made as to the timing of the creation and deletion of servant instances.

# 3.3 Transactions

The following sections provide information about transaction policies and how to use transactions.

- Transaction Policies
- Transaction Initiation
- Transaction Termination
- Transaction Suspend and Resume
- Restrictions on Transactions
- SQL and Global Transactions
- Voting on Transaction Outcome
- Transaction Timeouts

# 3.3.1 Transaction Policies

Eligibility of CORBA objects to participate in global transactions is controlled by the transaction policies assigned to implementations at compile time. The following policies can be assigned.



### Note:

The transaction policies are set in an ICF file that is configured at OMG IDL compile time. For a description of the ICF file, refer to the Implementation Configuration File (ICF) section.

#### never

The implementation is not transactional. Objects created for this interface can never be involved in a transaction. The system generates an exception (INVALID\_TRANSACTION) if an implementation with this policy is involved in a transaction. An AUTOTRAN policy specified in the UBBCONFIG file for the interface is ignored.

#### ignore

The implementation is not transactional. This policy instructs the system to allow requests within a transaction to be made of this implementation. An AUTOTRAN policy specified in the UBBCONFIG file for the interface is ignored.

optional (This is the default transaction\_policy.)
 The implementation may be transactional. Objects can be involved in a transaction if the request is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant resource manager. If the AUTOTRAN parameter is specified in the UBBCONFIG file for the interface, AUTOTRAN is on.

#### • always

The implementation is transactional. Objects are required to always be involved in a transaction. If a request is made outside a transaction, the system automatically starts a transaction before invoking the method. The transaction is committed when the method ends. (This is the same behavior that results from specifying AUTOTRAN for an object with the option transaction policy, except that no administrative configuration is necessary to achieve this behavior, and it cannot be overridden by administrative configuration.) Servers containing transactional objects must be configured within a group that is associated with an XA-compliant resource manager.

### Note:

The optional policy is the only transaction policy that can be influenced by administrative configuration. If the system administrator sets the AUTOTRAN attribute for the interface by means of the UBBCONFIG file or by using administrative tools, the system automatically starts a transaction upon invocation of the object, if it is not already infected with a transaction (that is, the behavior is as if the always policy were specified).

# 3.3.2 Transaction Initiation

Transactions are initiated in one of two ways:

- By the application code via use of the CosTransactions::Current::begin() operation. This can be done in either the client or the server. For a description of this operation, see Using CORBA Transactions.
- By the system when an invocation is done on an object which has either:
  - Transaction policy always



 Transaction policy optional and a setting of AUTOTRAN for the interface For more information, see *Using CORBA Transactions*.

# 3.3.3 Transaction Termination

In general, the handling of the outcome of a transaction is the responsibility of the initiator. Therefore, the following are true:

- If the client or server application code initiates transactions, the TP Framework never commits a transaction. The Oracle Tuxedo system may roll back the transaction if server processing tries to return to the client while the transaction is in an illegal state.
- If the system initiates a transaction, the commit or rollback will always be handled by the Oracle Tuxedo system.

The following behavior is enforced by the Oracle Tuxedo system:

• If no transaction is active when a method on a CORBA object is invoked and that method begins a transaction, the transaction must be either committed, rolled back, or suspended when the method invocation returns. If none of these actions is taken, the transaction is rolled back by the TP Framework, and the CORBA::OBJ\_ADAPTER exception is raised to the client application. This exception is raised because the transaction was initiated in the server application; therefore, the client application would not expect a transactional error condition such as TRANSACTION ROLLEDBACK.

# 3.3.4 Transaction Suspend and Resume

The CORBA object must follow strict rules with respect to suspending and resuming a transaction within a method invocation. These rules and the error conditions that result from their violation are described below.

When a CORBA object method begins execution, it can be in one of the following three states with respect to transactions:

- The client application began the transaction.
  - Legal server application behavior: Suspend and resume the transaction within the method execution.
  - Illegal server application behavior: Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
  - Error Processing: If illegal behavior occurs, the TP Framework raises the CORBA::TRANSACTION\_ROLLEDBACK exception to the client application and the transaction is rolled back by the Oracle Tuxedo system.
- The system began a transaction to provide AUTOTRAN or transaction policy always behavior.



For each CORBA interface, set AUTOTRAN to Yes if you want a transaction to start automatically when an operation invocation is received. Setting AUTOTRAN to Yes has no effect if the interface is already in transaction mode. For more information about AUTOTRAN, see *Using CORBA Transactions*.



 Legal server behavior: Suspend and resume the transaction within the method execution.



Not recommended. The transaction may be timed out and aborted before the method causes the transaction to be resumed.

- Illegal server behavior: Return from the method with the transaction in the suspended state (that is, return from the method without invoking resume if suspend was invoked).
- Error Processing: If illegal behavior occurs, the TP Framework raises the CORBA::OBJ\_ADAPTER exception to the client, and the transaction is rolled back by the system. The CORBA::OBJ\_ADAPTER exception is raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.
- The CORBA object is not involved in a transaction when it starts executing.
  - Legal server behavior:
    - \* Begin and commit a transaction within the method execution.
    - \* Begin and roll back a transaction within the method execution.
    - \* Begin and suspend a transaction within the method execution.
  - Illegal server behavior: Begin a transaction and return from the method with the transaction active.
  - Error Processing: If illegal behavior occurs, the TP Framework raises the CORBA::OBJ\_ADAPTER exception to the client application and the transaction is rolled back by the Oracle Tuxedo system. The CORBA::OBJ\_ADAPTER exception is raised because the client application did not initiate the transaction, and, therefore, does not expect transaction error conditions to be raised.

# 3.3.5 Restrictions on Transactions

The following restrictions apply to Oracle Tuxedo CORBA transactions:

- A CORBA object in the Oracle Tuxedo system must have the same transaction context when it returns from a method invocation that it had when the method was invoked.
- A CORBA object can be infected by only one transaction at a time. If an invocation tries to infect an already infected object, a CORBA::INVALID\_TRANSACTION exception is returned.
- If a CORBA object is infected with a transaction and a nontransactional request is made on it, a CORBA::OBJ ADAPTER exception is raised.
- If the application begins a transaction in Server::initialize(), it must either commit or roll back the transaction before returning from the method. If the application does not, the TP Framework shuts down the server. This is because the application has no predictable way of regaining control after completing the Server::initialize method.
- If a CORBA object is infected by a transaction and with an activation policy of transaction, and if the reason code passed to the method is either DR\_TRANS\_COMMITTING or DR\_TRANS\_ABORTED, no invocation on any CORBA object can be done from within the Tobj\_ServantBase::deactivate\_object method. Such an invocation results in a CORBA::BAD INV ORDER exception.



# 3.3.6 SQL and Global Transactions

Adhere to the following guidelines when using SQL and Global Transactions:

 Care should be taken when executing SQL statements outside the scope of a global transaction. The SQL standard specifies that a local transaction should be started implicitly by the database resource manager whenever an SQL statement that needs the context of a transaction is executed and no transaction is active. The standard also says that a transaction that is implicitly started by the database resource manager must then be explicitly terminated by executing a COMMIT or ROLLBACK SQL statement; the TP Framework is not responsible for terminating transactions that are started by the resource manager.

## Note:

This is not an issue when an application uses the XA library to connect to the Oracle server because those applications can operate only on global transactions. The Oracle server does not allow local transactions when it is using  $\chi_{\Delta}$ 

- The SQL COMMIT and ROLLBACK statements cannot be used to terminate a global transaction that has been either started explicitly using Current.begin() or started implicitly by the system. Check the database vendor documentation for each database product for other possible restrictions when using global transactions.
- SQL cursors may be closed when transactions are terminated. Consult your database
  product documentation for exact information about cursor handling rules. Application
  programmers should be careful to use cursors only with CORBA objects with appropriate
  activation policies and within appropriate transaction boundaries.

# 3.3.7 Voting on Transaction Outcome

CORBA objects can affect transaction outcome during two stages of transaction processing:

- During transactional work
   The Current.rollback\_only method can be used to ensure that the only possible outcome is to roll back the current transaction. Current.rollback\_only() can be invoked from any CORBA object method.
- After completion of transactional work
   CORBA objects that have the transaction activation policy are given a chance to vote
   whether the transaction should commit or roll back after transactional work is completed.
   These objects are notified of the completion of transactional work prior to the start of the
   two-phase commit algorithm when the TP Framework invokes their deactivate\_object
   method.

This behavior does not apply to objects with process or method activation policies. If the CORBA object wants to roll back the transaction, it can call Current::rollback\_only. If it wants to vote to commit the transaction, it does not make that call. Note, however, that a vote to commit does not guarantee that the transaction is committed, since other objects may subsequently vote to roll back the transaction.





Users of SQL cursors must be careful when using an object with the method or process activation policy. A process opens an SQL cursor within a client-initiated transaction. For typical SQL database products, once the client commits the transaction, all cursors that were opened within that transaction are automatically closed; however, the object will not receive any notification that its cursor has been closed.

# 3.3.8 Transaction Timeouts

When a transaction timeout occurs, the transaction is marked so that the only possible outcome is to roll back the transaction, and the <code>CORBA::TRANSACTION\_ROLLEDBACK</code> standard exception is returned to the client. Any attempts to send new requests will also fail with the <code>CORBA::TRANSACTION\_ROLLEDBACK</code> exception until the transaction has been aborted.

# 3.4 IIOP Client Failover

It is not always possible to determine when a server instance failed with respect to the work it was doing at the time of failure. For example, if a server instance fails after handling a client request but before returning the response, there is no way to tell that the request was handled. A user that does not get a response will most likely retry, resulting in an additional request.

Support for IIOP client failover has been added to Oracle Tuxedo CORBA as an availability enhancement. IIOP client failover provides a transparent mechanism for a CORBA remote client to automatically connect to an alternative ISL and then retry the request in case of failure.

IIOP client failover marks an interface implementation as *idempotent*. An idempotent implementation is one that can be repeated without any negative side-effects. For example, SET BALANCE.

- Setting The Retry Policy
- Initiating IIOP Client Failover

# 3.4.1 Setting The Retry Policy

To mark an interface implementation as idempotent, you must set the retry policy in the implementation configuration file (ICF) using the retry\_policy option. For a description of the ICF, refer to the Implementation Configuration File (ICF) section.

The retry policy option has two settings:

- never: The default setting. It indicates that the interface implementation is not idempotent and that requests should never be automatically retried.
- always: Indicates that the interface implementation is idempotent and that requests should always be retried in case of failure.
- MIB Support



# 3.4.1.1 MIB Support

You can also check the retry policy using the TA\_RTPOLICY attribute added to the MIB (5) T\_IFQUEUE and T\_INTERFACE classes. The TA\_RTPOLICY attribute value is either never or always.

# 3.4.2 Initiating IIOP Client Failover

To initiate IIOP client failover support, ISL servers must be specified using the -C warn|none option in the \*SERVERS section of the UBBCONFIG file

This option allows ISL to accept unofficial connection directly from the client orb. ISL servers that are not specified using the -C warn|none option will not be placed in candidate IIOP gateway pools. Consequently, the client will not failover to those ISL servers.

In the following UBBCONFIG file code snippet, the ISL servers specified in lines 1 and 2 must support client failover. The ISL server in line 3 will not.

```
*SERVERS

ISL SRVGRP=SYS_GRP1 SRVID=10 CLOPT="-A -- -C warn -n //myhost1:2468"

ISL SRVGRP=SYS_GRP2 SRVID=20 CLOPT="-A -- -C none -n //myhost2:2469"

ISL SRVGRP=SYS_GRP3 SRVID=30 CLOPT="-A -- -n //myhost3:2470"
```

IIOP Client Failover Limitations

### 3.4.2.1 IIOP Client Failover Limitations

IIOP Client Failover is not supported under the following three instances:

- Tuxedo system-supplied object failover
   Only application-supplied object failover is supported.
- Transaction mode
- SSL link or authentication is required

# See Also:

For more information about <code>UBBCONFIG(5)</code>, <code>MIB(5)</code>, <code>T\_IFQUEUE Class</code>, and <code>T\_INTERFACE Class</code> see in the Section 5 - File Formats, Data Descriptions, MIBs, and System Processes Reference.

# 3.5 WebLogic CORBA Clustering and Load Balancing Support

Tuxedo CORBA C++ client supports failover to WebLogic clustering servers and also supports load balancing. For more information, see WebLogic documentation - Failover and Replication in a Cluster and Load Balancing in a Cluster.



# 3.6 Parallel Objects

Support for parallel objects was added to Oracle Tuxedo CORBA in release 8.0 as a performance enhancement. The parallel objects feature enables you to designate all business objects in a particular application as stateless objects. The effect is that, unlike stateful business objects, which can only run on one server in a single domain, stateless business objects can run on all servers in a single domain. Thus, the benefits of parallel objects are as follows:

- Parallel objects can run on multiple servers in the same domain at the same time. Thus, utilization of all servers to service concurrent multiple requests improves performance.
- When the Oracle Tuxedo system services requests to parallel business objects, it always
  looks for an available server to the local machine first. If all servers on the local machine
  are busy processing the requested business object, the Oracle Tuxedo system looks for an
  available server on other machines in the local domain. Thus, if there are multiple servers
  on the local machine, network traffic is reduced and performance is improved.

For more information on parallel objects, see Using Parallel Objects.

To implement parallel objects, the concurrency policy option has been added to the ICF file. To select parallel objects for a particular application, you set the concurrency policy option to user-controlled. When you select user-controlled concurrency, the business object is not registered with the Active Object Map (AOM) and, therefore, are stateless and can be active on more than one server at a time. Thus, these objects are referred to as parallel objects.

If user-controlled concurrency is selected, the servant implementation must comply with one of the following statements:

- The servant implementation must have no requirements for concurrent access to a shared resource
- Or the servant implementation must utilize some other tool (for example, a database and locking) to ensure the correct behavior during concurrent access to resources.

In release 8.0 of the Oracle Tuxedo software, the Implementation Configuration File (ICF) was modified to support user-controlled concurrency. In following listing, the changes to add this support are highlighted in **bold** type. For a description of the ICF syntax, see ICF Syntax.

User-controlled concurrency can be used with factory-based routing, all activation policies, and all transaction policies. The interaction with these features is as follows:

- Factory-based routing
  - If the user specifies factory-based routing when creating the object, then the object will route to a server in that group. The object key contains the group selected during factory-based routing, but the client routing code will recognize that the interface has user-controlled concurrency and specify the desired group. This is accomplished using normal Oracle Tuxedo routing.
- Activation policy
  - The TP Framework handles active user-controlled concurrency objects in the same manner as system-controlled concurrency objects. The TP Framework stores information about objects in the local AOM, and calls the activate\_object and deactivate\_object methods at the appropriate times. However, the object will not have an entry in the AOM and the TP Framework will not call any AOM routines. For example, on shutdown, since an active object will not have an AOM handle, calls to remove the entry from the AOM will not be invoked.
- Transaction policy

The TP Framework handles active user-controlled concurrency objects in the same manner as system-controlled concurrency objects. The TP Framework is called back for transaction events and the TP Framework stores information about transactional user-controlled objects in the local AOM. The main differences when using parallel objects in transactions as opposed to stateful objects are that the AOM is not used for GTRID information and the AOM routines are not called to update or retrieve transactional information.

## Note:

There is one restriction with user-controlled concurrency.

TP::create\_active\_object\_reference throws a TobjS::IllegalOperation exception if it is passed an interface with user-controlled concurrency set. Since the AOM is not used when user-controlled concurrency is set, there is no way for the TP Framework to connect an active object to this server.

# 3.7 TP Framework API

This section describes the TP Framework API. Additional information about how to use this API can be found in *Creating CORBA Server Applications*.

The TP Framework comprises the following components:

- The Server C++ class, which has virtual methods for application-specific server initialization and termination logic
- The ServerBase C++ class, which has virtual methods for multithreaded server applications.
- The Tobj ServantBase C++ class, which has virtual methods for object state management
- The TP C++ class, which provides methods to:
  - Create object references for CORBA objects
  - Register (and unregister) factories with the FactoryFinder object
  - Initiate user-controlled preactivation and deactivation of objects
  - Initiate user-controlled deactivation of the CORBA object currently being invoked
  - Obtain an object reference to the CORBA object currently being invoked

- Open and close XA resource managers
- Log messages to a user log (ULOG) file
- Obtain object references to the ORB and to Bootstrap objects (if not using the CORBA Interoperable Naming Service (INS))
- Header files for these classes
- Libraries that are used by server applications
   The visible part of the TP Framework consists of two categories of operations:
- Service methods that can be called by user code. These are in the TP interface.
- Callback methods that are written by the user and that are invoked by the TP Framework.
   This includes methods in the Tobj\_ServantBase and Server classes. These operations are intended to be called by TP Framework code only. The application code should never call the methods of these classes. If it does, unpredictable results may occur.
- Server Interface
- Tobj\_ServantBase Interface
- TP Interface

# 3.7.1 Server Interface

The Server interface provides callback methods that can be used for application-specific server initialization and termination logic. This interface also provides a callback method that is used to create servants when servants are required for object activation.

The Server interface has the following characteristics:

- The Server class inherits from the ServerBase class.
- The Server class is a C++ native class.
- The Server.h file contains the declarations and definitions for the Server class.

For a description of the Server interface methods, see ServerBase Interface.

- C++ Declarations
- Server::create servant()
- Server::initialize()
- Server::release()

### 3.7.1.1 C++ Declarations

For the C++ mappings, see ServerBase Interface.

ServerBase Interface

### 3.7.1.1.1 ServerBase Interface

The serverBase interface allows you to take full advantage of multithreading capabilities. You can create your own Server classes that inherit from the ServerBase class. This provides you with the following:

The create\_servant\_with\_id() method to support implementations requiring knowledge
of the target object during the creation of a servant



Support for user-supplied thread initialization and release handlers

The ServerBase class provides the same operations that were available in the Server class in earlier releases. The Server class inherits from the ServerBase class.

These methods can be used with single-threaded and multithreaded applications:

```
Server::create_servant()
Server::initialize()
Server::release()
Server::initialize()
Server::release()
ServerBase::create servant with id()
```

These methods can be used with multithreaded server applications only:

```
• ServerBase:: thread_initialize()
```

• ServerBase::thread release()



Programmers must provide definitions of the Server class methods. The ServerBase class methods have default implementations.

C++ Declarations (in Server.h)

### 3.7.1.1.1 C++ Declarations (in Server.h)

The C++ mapping is as follows:

```
class OBBEXPDLLUSER ServerBase {
public:
    virtual CORBA::Boolean
        initialize(int argc, char ** argv) = 0;
   virtual void
       release() = 0;
   virtual Tobj Servant
       create servant(const char * interfaceName) = 0;
       // Default Implementations Supplied
    virtual Tobj Servant
       create servant with id(const char * interfaceName,
                             const char * stroid);
    virtual CORBA::Boolean
       thread initialize(int argc, char ** argv);
   virtual void
       thread release();
};
class Server: public ServerBase {
   public: CORBA::Boolean initialize(int argc, char ** argv);
   void release();
   Tobj Servant create servant(const char * interfaceName);
};
```

# 3.7.1.2 Server::create servant()

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Value
- ServerBase::create servant with id()

## 3.7.1.2.1 Synopsis

Creates a servant to instantiate a C++ object.

## 3.7.1.2.2 C++ Binding

```
class Server {
    public:
        Tobj_Servant create_servant(const char * interfaceName);
};
```

## 3.7.1.2.3 Argument

#### interfaceName

Specifies a character string that contains the fully qualified interface name for the object. This will be the same interface name that was supplied when the object reference was created (TP::create\_object\_reference() or TP::create\_active\_object\_reference()) for the object reference used for this invocation. This name can be used to determine which servant needs to be constructed.

# 3.7.1.2.4 Exception

If an exception is thrown in <code>Server::create\_servant()</code>, the TP Framework catches the exception. Activation fails. A <code>CORBA::OBJECT\_NOT\_EXIST()</code> exception is raised back to the client. In addition, an error message is written to the user log (<code>ULOG</code>) file, as follows, for each exception type:

#### TobjS::CreateServantFailed

```
"TPFW_CAT:23: ERROR: Activating object - application raised
TobjS::CreateServantFailed. Reason = reason. Interface = interfaceName, OID = oid
"
```

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

#### TobjS::OutOfMemory

```
"TPFW_CAT:22: ERROR: Activating object - application raised TobjS::OutOfMemory. Reason = reason. Interface = interfaceName , OID = oid "
```



Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

#### CORBA:: Exception

```
"TPFW_CAT:28: ERROR: Activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName , OID = oid "
```

#### CORBA:: Exception

```
"TPFW_CAT:28: ERROR: Activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName , OID = oid "
```

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

#### Other Exception

```
"TPFW_CAT:29: ERROR: Activating object - Unknown Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName , OID = oid "
```

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

## 3.7.1.2.5 Description

The create\_servant method is invoked by the TP Framework when a request arrives at the server and there is no available servant to satisfy the request. The TP Framework calls the create\_servant method with the interface name for the servant to be created. The server application instantiates an appropriate C++ object and returns a pointer to it. Typically, the method contains a switch statement on the interface name and creates a new object, depending on the interface name.



### **Caution:**

The server application must not depend on this method being invoked for every activation of a CORBA object. The server application must not do any handling of CORBA object state in the constructors or destructors of any servant classes for CORBA objects. This is because the TP Framework may possibly reuse servants on activation and may possibly not destroy servants on deactivation.

### 3.7.1.2.6 Return Value

#### Tobi Servant

The pointer to the newly created servant (instance) for the specified interface. A NULL value should be returned if <code>create\_servant()</code> is invoked with an interface name that it does not recognize or if the servant cannot be created for some reason.

The pointer to the newly created servant (instance) for the specified interface. A NULL value should be returned if <code>create\_servant()</code> is invoked with an interface name that it does not recognize or if the servant cannot be created for some reason.

```
"TPFW_CAT:23: ERROR: Activating object - application raised
TobjS::CreateServantFailed. Reason = Application's Server::create_servant
returned NULL. Interface = interfaceName , OID = oid "
```



Where <code>interfaceName</code> is the interface ID of the invoked interface and <code>oid</code> is the corresponding object ID.



The restriction on the length of the <code>ObjectId</code> has been removed in this release.

## 3.7.1.2.7 ServerBase::create servant with id()

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Value
- Example

### 3.7.1.2.7.1 Synopsis

Creates a servant for this target object. This method supports the development of single-headed and multithreaded server applications.

### 3.7.1.2.7.2 C++ Binding

```
Tobj_Servant create_servant_with_id (const char* interfaceName,
const char* stroid);
```

### 3.7.1.2.7.3 Arguments

#### interfaceName

Specifies a character string containing the fully qualified interface name for the object. This must be the same interface name that was supplied when the object reference was created.

#### stroid

Specifies an object ID in string format. The object ID uniquely identifies the object associated with the request to be processed. This is the same object ID that was specified when the object reference was created.

#### 3.7.1.2.7.4 Description

The TP Framework invokes the <code>create\_servant\_with\_id</code> method when a request arrives at the server and there no servant is available to satisfy the request. The TP Framework passes in the interface name for the servant to be created and the object ID associated with the object with which the servant will be associated. The server application instantiates an appropriate C+ + object and returns a pointer to it. Typically, the method contains a <code>switch</code> statement on the interface name and creates a new object, depending on the interface name. Providing the object ID allows a servant implementation to make decisions during the creation of the servant instance that require knowledge of the target object. Reentrancy support is one example of how a servant implementation might employ knowledge of the target object.

The ServerBase class provides a default implementation of create\_servant\_with\_id which calls the standard create\_servant method passing the interface name. This default implementation ignores the target object ID parameter.



#### Caution:

The server application must not depend on the invocation of this method for every activation of a CORBA object. The server application must not handle the CORBA object state in the constructors or destructors of any servant classes for CORBA objects. This is because the TP Framework might reuse servants on activation and might not destroy servants on deactivation.

#### 3.7.1.2.7.5 Return Value

#### Tobj Servant

A pointer to the newly created servant (instance) for the specified interface. Returns  $\mathtt{NULL}$  if either of these conditions is true:

- Interface name not recognized
- Unable to create a servant

### 3.7.1.2.7.6 Example

# 3.7.1.3 Server::initialize()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value
- ServerBase::thread\_initialize()

# 3.7.1.3.1 Synopsis

Allows the application to perform application-specific initialization procedures, such as logging into a database, creating and registering well-known object factories, initializing global variables, and so forth.

# 3.7.1.3.2 C++ Binding

## 3.7.1.3.3 Arguments

The argc and argv arguments are passed from the command line. The argc argument contains the name of the server. The argv argument contains the first command-line option that is specific to the application, if there are any.

Command-line options are specified in the UBBCONFIG file using the CLOPT parameter in the entry for the server in the SERVERS section. System-recognized options come first in the CLOPT parameter, followed by a double-hyphen (--), followed by the application-specific options. The value of argc is one greater than the number of application-specific options. For details, see ubbconfig (5) in the File Formats, Data Descriptions, MIBs, and System Processes Reference.

# **3.7.1.3.4 Exceptions**

If an exception is raised in Server::initialize(), the TP Framework catches the exception. The TP Framework behavior is the same as if initialize() returned FALSE (that is, an exception is considered to be a failure). In addition, an error message is written to the user log (ULOG) file, as follows, for each exception type:

```
TobjS::InitializeFailed
```

```
"TPFW_CAT:1: ERROR: Exception in Server::initialize():IDL:beasys.com/TobjS/InitializeFailed:1.0. Reason = reason "
```

Where reason is a string supplied by application code. For example: Throw TobjS::InitializeFailed("Couldn't register factory");

#### CORBA::Exception

```
"TPFW_CAT:1: ERROR: Exception in Server::initialize(): exception . Reason = unknown"
```

Where exception is the interface ID of the CORBA exception that was raised.

#### Other Exceptions

```
TPFW_CAT:1: ERROR: Exception in Server::initialize(): unknown exception. Reason
= unknown"
```

# 3.7.1.3.5 Description

The initialize callback method, which is invoked as the last step in server initialization, allows the application to perform application-specific initialization.

Typically, a server application does the following tasks in Server::initialize:

- Creates references for CORBA object factories implemented in the server application and registers them with the FactoryFinder using the TP::register factory() operation.
- Initializes global variables, if any are used.

Opens XA resource managers if any are used by the server application.

It is the responsibility of the server application to open any required XA resource managers. This is done by invoking either of the following methods:

• TP::open\_xa\_rm() This is the preferred technique for server applications, since it can be done on a static function, without the need to obtain an object reference.

# Note:

You must use the  $TP::open_xa_rm()$  method if you use the INS bootstrap mechanism to obtain initial object references.

- Tobj::TransactionCurrent::open\_xa\_rm() A reference to the TransactionCurrent object
  can be obtained from the Bootstrap object. For an explanation of how to obtain a reference
  to the Bootstrap object, see the section TP::bootstrap(). For more information about the
  TransactionCurrent object, see the CORBA Bootstrapping Programming Reference section
  and Using CORBA Transactions.
- Transactions may be started in the initialize method after invoking the Tobj::TransactionCurrent::open\_xa\_rm() or TP::open\_xa\_rm method. However, any transactions that are started in initialize() must be terminated by the server application before initialize() returns. If the transactions are still active when control is returned, the server application fails to boot, and it exits gracefully. This happens because the server application has no logical way of either committing or rolling back the transaction after Server::initialize() returns. This condition is an error.

#### 3.7.1.3.6 Return Value

Boolean TRUE or FALSE. TRUE indicates success. FALSE indicates failure. If an error occurs in initialize(), the application code should return FALSE. The application code should not call the system call exit(). Calling exit() does not give the TP Framework a chance to release resources allocated during startup and may cause unpredictable results.

If the return value is FALSE:

- Server::release() is not invoked.
- Any transactions that are started in the initialize() method and are not terminated will eventually time out; they are not automatically rolled back.

## 3.7.1.3.7 ServerBase::thread initialize()

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Value
- Example

#### 3.7.1.3.7.1 Synopsis

Performs any necessary application-specific initialization for a thread created using the Oracle Tuxedo software. This method supports the development of a multithreaded server application.

### 3.7.1.3.7.2 C++ Binding

CORBA::Boolean thread initialize(int argc, char\*\* argv)

### 3.7.1.3.7.3 Arguments

#### argc

The number of arguments provided to the application. Initially, this count is passed to the main function.

#### argv

The arguments provided to the application. Initially, these arguments are passed to the main function.

### 3.7.1.3.7.4 Description

In managing the thread pool, the Oracle Tuxedo software creates and releases threads using the operating system thread library services. Depending on application requirements, these threads might need to be initialized before they are used to process requests.

The thread\_initialize callback method is invoked each time a thread is created, to initialize the thread. Note that the Oracle Tuxedo software manages a number of system-owned threads that are used for dispatching requests; these system-owned threads are in addition to those threads in the thread pool. Under some circumstances the servant methods you implement are also executed in these system-owned threads; for this reason the Oracle Tuxedo software invokes the thread initialize method to initialize the system-owned threads.

The ServerBase class provides a default implementation of the thread\_initialize method that opens the XA resource manager in the initialized thread.

#### 3.7.1.3.7.5 Return Value

#### CORBA::Boolean

True if the initialization of the thread was successful.

#### 3.7.1.3.7.6 Example

# 3.7.1.4 Server::release()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description



- Return Value
- ServerBase::thread release()

## 3.7.1.4.1 Synopsis

Allows the application to perform any application-specific cleanup, such as logging off a database, unregistering well-known factories, or deallocating resources.

## 3.7.1.4.2 C++ Binding

```
typedef Tobj_ServantBase * Tobj_Servant;
class Server {
         public: void release();
};
```

# 3.7.1.4.3 Arguments

None.

## 3.7.1.4.4 Exceptions

If an exception is raised in release(), the TP Framework catches the exception. Each exception causes an error message to be written to the user log (ULOG) file, as follows:

#### TobjS::ReleaseFailed

```
"TPFW_CAT:2: WARN: Exception in Server::release():
IDL:beasys.com/TobjS/ReleaseFailed:1.0. Reason = reason"
```

Where reason is a string supplied by application code. For example:

```
Throw TobjS::ReleaseFailed("Couldn't unregister factory");
```

#### CORBA:: Exception

```
"TPFW CAT:2: WARN: Exception in Server::release(): exception . Reason = unknown"
```

Where exception is the interface ID of the CORBA exception that was raised.

#### Other Exceptions

```
"TPFW_CAT:2: WARN: Exception in Server::release(): unknown exception. Reason = unknown"
```

In all cases, the server continues to exit.

### 3.7.1.4.5 Description

The release callback method, which is invoked as the first step in server shutdown, allows the server application to perform any application-specific cleanup. The user must override the virtual function definition.

Typical tasks performed by the application in this method are as follows:

- Close XA resource managers.
- Unregister CORBA object factories that were registered with the FactoryFinder in Server::initialize().
- Deallocate any server resources not yet released.

This method is normally called in response to a tmshutdown command from the administrator or operator.

The TP Framework provides a default implementation of Server::release(). The default implementation closes XA resource managers for the server. The implementation does this by issuing a  $tx\_close()$  invocation, which uses the default CLOSEINFO configured for the server's group in the UBBCONFIG file.

It is the responsibility of the application to close any open XA resource managers. This is done by issuing either of the following calls:

• TP::close xa rm()

### Note:

You must use the  $\mathtt{TP::close\_xa\_rm}()$  method if you use the INS bootstrap mechanism to obtain initial object references.

Tobj::TransactionCurrent::close\_xa\_rm(). A reference to the TransactionCurrent object can be obtained from the Bootstrap object. For an explanation of how to obtain a reference to the Bootstrap object, see the section TP::bootstrap(). For more information about the TransactionCurrent object, see CORBA Bootstrapping Programming Reference and Using CORBA Transactions.

#### Note:

Once a server receives a request from the tmshutdown(1) command to shut down, it can no longer receive requests from other remote objects. This may require servers to be shut down in a specific order. For example, if the Server::release() method in Server 1 needs to access a method of an object that resides in Server 2, Server 2 should be shut down after Sever 1 is shut down. In particular, the TP::unregister\_factory() method accesses the FactoryFinder Registrar object that resides in a separate server. The TP::unregister\_factory() method is typically invoked from the release() method; therefore, the FactoryFinder server should be shut down after all servers that call TP::unregister\_factory() in their Server::release() method.

### 3.7.1.4.6 Return Value

None.

### 3.7.1.4.7 ServerBase::thread release()

- Synopsis
- C++ Binding
- Arguments



- Description
- Return Value
- Example

### 3.7.1.4.7.1 Synopsis

Performs application-specific cleanup when a thread that was created by the Oracle Tuxedo software is released. This method supports the development of a multithreaded server application.

#### 3.7.1.4.7.2 C++ Binding

```
void thread release()
```

### 3.7.1.4.7.3 Arguments

None.

### 3.7.1.4.7.4 Description

The thread\_release callback method is invoked each time a thread is released. Implement the thread\_release method as necessary to perform application-specific resource cleanup.

The ServerBase class provides a default implementation of the thread\_release method that closes the XA resource manager in the released thread.

#### 3.7.1.4.7.5 Return Value

None.

#### 3.7.1.4.7.6 Example

```
void simple_per_request_server::thread_release()
{
TP::userlog("thread_release called in thread %ld",
(unsigned long)SIMPTHR_GETCURRENTTHREADID);
}
```

# 3.7.2 Tobj\_ServantBase Interface

The Tobj\_ServantBase interface inherits from the PortableServer::RefCountServantBase class and defines operations that allow a CORBA object to assist in the management of its state in a thread-safe manner. Every implementation skeleton generated by the IDL compiler automatically inherits from the Tobj\_ServantBase class. The Tobj\_ServantBase class contains two virtual methods, activate\_object() and deactivate\_object(), that may be optionally implemented by the programmer.

Whenever a request comes in for an inactive CORBA object, the object is activated and the activate\_object() method is invoked on the servant. When the CORBA object is deactivated, the deactivate\_object() method is invoked on the servant. The timing of deactivation is driven by the implementation's activation policy. When the deactivate\_object() method is invoked, the TP Framework passes in a reason code to indicate why the call was made.

These methods support the development of a multithreaded server application:

- TobjServantBase:: add ref()
- TobjServantBase:: is reentrant()
- TobjServantBase:: remove ref()

#### Note:

Tobj\_ServantBase::activate\_object() and
Tobj\_ServantBase::deactivate\_object() are the only methods that the TP
Framework guarantees will be invoked for CORBA object activation and
deactivation. The servant class constructor and destructor may or may not be
invoked at activation or deactivation time (through the Server::create\_servant
call for C++). Therefore, the server application code must not do any state
handling for CORBA objects in either the constructor or destructor of the servant

The programmer does not need to use a cast or reference to <code>Tobj\_ServantBase</code> directly. The <code>Tobj\_ServantBase</code> methods show up as part of the skeleton and, therefore, in the implementation class for a servant. The programmer may provide definitions for the <code>activate\_object</code> and <code>deactivate\_object</code> methods, but the programmer should never make direct invocations on those methods; only the TP Framework should call those methods.

- C++ Declaration (in Tobj\_ServantBase.h)
- Tobj ServantBase:: activate object()
- Tobj\_ServantBase::\_add\_ref()
- Tobj\_ServantBase::deactivate\_object()
- Tobj\_ServantBase::\_is\_reentrant()
- Tobj\_ServantBase::\_remove\_ref()

# 3.7.2.1 C++ Declaration (in Tobj ServantBase.h)

The C++ mapping for the Tobj servantBase interface is as follows:

```
class Tobj_ServantBase : public PortableServer::RefCountServantBase {
  public:
  Tobj_ServantBase& operator=(const Tobj_ServantBase&);
  Tobj_ServantBase() {}
  Tobj_ServantBase(const Tobj_ServantBase& s) :
  PortableServer::RefCountServantBase(s) {}
  virtual void activate_object(const char *) {}
  virtual void deactivate_object(const char*,
  TobjS::DeactivateReasonValue) {}
  virtual CORBA::Boolean _is_reentrant() { return CORBA_FALSE; }
  };
  typedef Tobj_ServantBase * Tobj_Servant;
```

# 3.7.2.2 Tobj ServantBase:: activate object()

Synopsis

- C++ Binding
- Argument
- Description
- Return Value
- Exceptions

# 3.7.2.2.1 Synopsis

Associates an object ID with a servant. This method gives the application an opportunity to restore the object's state when the object is activated. The state may be restored from shared memory, from an ordinary flat file, or from a database file.

### 3.7.2.2.2 C++ Binding

```
class Tobj_ServantBase : public PortableServer::ServantBase {
  public:
  virtual void activate_object(const char * stroid) {}
};
```

### 3.7.2.2.3 Argument

#### stroid

Specifies the object ID in string format. The object ID uniquely identifies this instance of the class. This is the same object ID that was specified when the object reference was created (using TP:create\_object\_reference()) or in the TP::create\_active\_object\_reference() for the object reference used for this invocation.



The restriction on the length of the object ID has been removed in this release.

# 3.7.2.2.4 Description

Object activation is triggered by a client invoking a method on an inactive CORBA object. This causes the Portable Object Adapter (POA) to assign a servant to the CORBA object. The activate\_object() method is invoked before the method invoked by the client is invoked. If activate\_object() returns successfully, that is, without raising an exception, the requested method is executed on the servant.

The activate\_object() and deactivate\_object() methods and the method invoked by the client can be used by the programmer to manage object state. The particular way these methods are used to manage object state may vary according to the needs of the application. For a discussion of how these methods might be used, see *Creating CORBA Server Applications*.

If the object is currently infected with a global transaction, <code>activate\_object()</code> executes within the scope of that same global transaction.

It is the responsibility of the programmer of the object to check that the stored state of the object is consistent. In other words, it is up to the application code to save a persistent flag that

indicates whether or not deactivate\_object() successfully saved the state of the object. That flag should be checked in activate object().

#### 3.7.2.2.5 Return Value

None.

### 3.7.2.2.6 Exceptions

If an error occurs while executing <code>activate\_object()</code>, the application code should raise either a CORBA standard exception or a <code>TobjS::ActivateObjectFailed</code> exception. When an exception is raised, the TP Framework catches the exception, and the following events occur:

- The activation fails.
- The method invoked by the client is not executed.
- If activate\_object() is executing within a transaction and the client initiated the transaction, the transaction is *not* rolled back.
- A CORBA::OBJECT\_NOT\_EXIST exception is raised back to the client.



For each CORBA interface, set AUTOTRAN to Yes if you want a transaction to start automatically when an operation invocation is received. Setting AUTOTRAN to Yes has no effect if the interface is already in transaction mode. For more information about AUTOTRAN, see *Using CORBA Transactions*.

 Based on the exception is raised, a message is written to the user log (ULOG) file, as follows:

#### TobjS::ActivateObjectFailed

```
"TPFW_CAT:24: ERROR: Activating object - application raised TobjS::ActivateObjectFailed. Reason = reason . Interface = interfaceName , OID = oid "
```

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

#### TobjS::OutOfMemory

```
"TPFW_CAT:22: ERROR: Activating object - application raised TobjS::OutOfMemory. Reason = reason. Interface = interfaceName, OID = oid"
```

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.

#### CORBA:: Exception

"TPFW\_CAT:25: ERROR: Activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName, OID = oid"

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

#### Other exception

"TPFW\_CAT:26: ERROR: Activating object - Unknown Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName, OID = oid"

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

# 3.7.2.3 Tobj ServantBase:: add ref()

- Synopsis
- · C++ Binding
- Arguments
- Description
- Return Value
- Example

### 3.7.2.3.1 Synopsis

Adds a reference to a servant. This method supports the development of a multithreaded server application.



In applications written using Oracle Tuxedo release 8.0 or later, use this method instead of the TP::application responsibility() method.

#### 3.7.2.3.2 C++ Binding

void add ref()

### 3.7.2.3.3 Arguments

None.

# 3.7.2.3.4 Description

Invoke this method when a reference to a servant is needed. Invoking this method causes the reference count for the servant to increment by one.

#### 3.7.2.3.5 Return Value

None.

### 3.7.2.3.6 Example

```
myServant * servant = new intf_i();
if(servant != NULL)
servant-> add ref();
```

# 3.7.2.4 Tobj\_ServantBase::deactivate\_object()

- Synopsis
- C++ Binding
- Arguments
- Description
- Restriction
- Return Value
- Exceptions

# 3.7.2.4.1 Synopsis

Removes the association of an object ID with its servant. This method gives the application an opportunity to save all or part of the object's state before the object is deactivated. The state may be saved in shared memory, in an ordinary flat file, or in a database file.

# 3.7.2.4.2 C++ Binding

```
class Tobj_ServantBase : public PortableServer::ServantBase {
  public:
    virtual void deactivate_object(const char* stroid,
    TobjS::DeactivateReasonValue reason) {}
};
```

# 3.7.2.4.3 Arguments

#### stroid

Specifies the object ID in string format. The object ID uniquely identifies this instance of the class.



The restriction on the length of the object ID has been removed in this release.

#### reason

Indicates the event that caused this method to be invoked. The reason code can be one of the following:

#### DR METHOD END

Indicates that the object is being deactivated after the completion of a method. It is used if the object's deactivation policy is:

- method
- transaction (only if there is no transaction in effect)
- process (if TP::deactivateEnable() called)

#### DR SERVER SHUTDOWN

Indicates that the object is being deactivated because the server is being shut down in an orderly fashion. It is used if the object's deactivation policy is:

- transaction (only if transaction is active)
- process



When a server is shut down in an orderly fashion, all transactions that the server is involved in are marked for rollback.

#### DR TRANS ABORTED

This reason code is used only for objects that have the transaction activation policy. It can occur when the transaction is started by either the client or automatically by the system. When the <code>deactivate\_object()</code> method is invoked with this reason code, the transaction is marked for rollback only.

#### DR\_TRANS\_COMMITTING

This reason code is used only for objects that have the transaction activation policy. It can occur when the transaction is started by either the client or the TP Framework. It indicates that a <code>Current.commit()</code> operation was invoked for the transaction in which the object is involved. The <code>deactivate\_object()</code> method is invoked just before the transaction manager's two-phase commit algorithm begins; that is, before <code>prepare</code> is sent to the resource managers.

The CORBA object is allowed to vote on the outcome of the transaction when the deactivate\_object() method is invoked with the DR\_TRANS\_COMMITTING reason code. By invoking Current.rollback\_only(), the method can force the transaction to be rolled back; otherwise, the two-phase commit algorithm continues. The transaction is not necessarily committed just because the Current.rollback\_only() is not invoked in this method. Any other CORBA object or resource manager involved in the transaction could also vote to roll back the transaction.

#### DR\_EXPLICIT\_DEACTIVATE

Indicates that the object is being deactivated because the application executed a  $\protect\operatorname{TP::deactivateEnable}(-,-,-)$  on this object. This can happen only for objects that have the process activation policy.

# 3.7.2.4.4 Description

Object deactivation is initiated either by the system or by the application, depending on the activation policy of the implementation for the CORBA object. The deactivate\_object() method is invoked before the CORBA object is deactivated. For details of these policies and their use, see the section ICF Syntax.



Deactivation may occur after an execution of a method invoked by a client if the activation policy for the CORBA object implementation is method, or as a result of the end of transactional work if the activation policy is transaction. It may also occur as the result of server shutdown if the activation policy is transaction or process.

#### Note:

The deactivate\_object method will be called at server shutdown time for every object remaining in the Active Object Map, whether it was entered there implicitly by the TP Framework (the activation-on-demand technique: TP::create\_servant and the servant's activate\_object method) or explicitly by the user with TP::create\_active\_object\_reference.

The activate\_object() and deactivate\_object() methods and explicit methods invoked by the client can be used by the programmer to manage object state. The manner in which these methods are used to manage object state may vary according to the needs of the application. For a discussion of how these methods might be used, see *Creating CORBA Server Applications*.

The CORBA object with transaction activation policy gets to vote on the outcome of the transaction when the <code>deactivate\_object()</code> method is invoked with the <code>DR\_TRANS\_COMMITTING</code> reason code. By calling <code>Current.rollback\_only()</code> the method can force the transaction to be rolled back; otherwise, the two-phase commit algorithm continues. The transaction will not necessarily be committed just because <code>Current.rollback\_only()</code> is not called in this method. Any other CORBA object or resource manager involved in the transaction could also vote to roll back the transaction.

#### 3.7.2.4.5 Restriction

If the object is involved in a transaction when this method is invoked, there are restrictions on what type of processing can be done based on the reason the object is invoked. If the object was involved in a transaction, the activation policy is transaction and the reason code for the call is:

#### DR TRANS ABORTED

No invocations on any CORBA objects are allowed in the method. No tpcall() is allowed. Transactions cannot be suspended or begun.

#### DR TRANS COMMITTING

No invocations on any CORBA objects are allowed in the method. No tpcall() is allowed. Transactions cannot be suspended or begun.

The reason for these restrictions is that the deactivation of objects with activation policy transaction is controlled by a call to the TP Framework from the transaction manager for the

transaction. When the call with reason code <code>DR\_TRANS\_COMMITTING</code> is made, the transaction manager is executing phase 1 (prepare) of the two-phase commit. At this stage, it is not possible to issue a call to suspend a transaction nor to initiate a new transaction. Since a call to a CORBA object that was in another process would require that process to join the transaction, and the transaction manager is already executing the prepare phase, this would cause an error  $^1$ . Since a call to a CORBA object that had no transactional properties would require that the current transaction be suspended, this would also cause an error. The same is true of a <code>tpcall()</code>.

Similarly, when the invocation with reason code DR\_TRANS\_ABORTED is made, the transaction manager is already aborting. While the transaction manager is aborting, it is not possible to either suspend a transaction or initiate a new transaction. The same restrictions apply as for DR\_TRANS\_COMMITTING.

#### 3.7.2.4.6 Return Value

None.

### 3.7.2.4.7 Exceptions

If the CORBA object method that is invoked by the client raises an exception, that exception is caught by the TP Framework and is eventually returned to the client. This is true even if deactivate object() is invoked and raises an exception.

The client will never be notified about exceptions that are raised in deactivate\_object(). It is the responsibility of the application code to check that the stored state of the CORBA object is consistent. For example, the application code could save a persistent flag that indicates whether or not deactivate\_object() successfully saved the state. That flag can then be checked in activate object().

If an error occurs while executing <code>deactivate\_object()</code>, the application code should raise either a CORBA standard exception or a <code>DeactivateObjectFailed</code> exception If <code>deactivate\_object()</code> was invoked by the TP Framework, the TP Framework catches the exception and the following events occur:

- The object is deactivated.
- If the client initiated a transaction, the transaction is not rolled back.
- The client is not notified of the exception that is raised in <code>deactivate\_object()</code>.
- Based on which exception is raised, a message is logged to the user log (ULOG) file, as follows:

#### TobjS::DeactivateObjectFailed

```
"TPFW_CAT:27: ERROR: De-activating object - application raised TobjS::DeactivateObjectFailed. Reason = reason . Interface = interfaceName , OID = oid "
```

Where reason is a user-supplied reason, and interfaceName and oid are the interface ID and object ID, respectively, of the invoked CORBA object.



#### CORBA:: Exception

```
"TPFW_CAT:28: ERROR: De-activating object - CORBA Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName, OID = oid"
```

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

#### Other exception

```
"TPFW_CAT:29: ERROR: De-activating object - Unknown Exception not handled by application. Exception ID = exceptionID. Interface = interfaceName, OID = oid"
```

Where <code>exceptionID</code> is the interface ID of the exception, and <code>interfaceName</code> and <code>oid</code> are the interface ID and object ID, respectively, of the invoked CORBA object.

# 3.7.2.5 Tobj ServantBase:: is reentrant()

- Synopsis
- · C++ Binding
- Arguments
- Description
- Return Value
- Example

# 3.7.2.5.1 Synopsis

Indicates that the object supports concurrent, reentrant invocations. This method supports the development of a multithreaded server application.

### 3.7.2.5.2 C++ Binding

```
CORBA::Boolean _is_reentrant()
```

# 3.7.2.5.3 Arguments

None.

# 3.7.2.5.4 Description

The Oracle Tuxedo server infrastructure calls this method to determine whether the servant implementation supports a reentrant invocation. To support reentrancy, a servant must include the necessary code to protect the integrity of its state while multiple threads interact with the object.

The Tobj\_ServantBase class provides a default implementation of the \_is\_reentrant method that returns FALSE.

#### 3.7.2.5.5 Return Value

#### CORBA::Boolean

Returns TRUE if the servant can support reentrancy.

# 3.7.2.5.6 Example

```
CORBA::Boolean Simple_i::_is_reentrant()
{ TP::userlog("_is_reentrant called in thread %ld",
  (unsigned long)SIMPTHR_GETCURRENTTHREADID);
return CORBA_TRUE;
}
```

# 3.7.2.6 Tobj ServantBase:: remove ref()

- Synopsis
- C++ Binding
- Parameters
- Description
- Return Value
- Example

# 3.7.2.6.1 Synopsis

Releases a reference to a servant. This method supports the development of a multithreaded server application.



In applications written using Oracle Tuxedo release 8.0 or later, use this method instead of the C++ "delete" statement that you used previously with the TP::application\_responsibility() method.

# 3.7.2.6.2 C++ Binding

```
void _remove_ref()
```

#### 3.7.2.6.3 Parameters

None.

# 3.7.2.6.4 Description

Invoke this method when a reference to a servant is no longer needed. Invoking this method causes the reference count for the servant to be decremented by one. If the \_remove\_ref() method brings the reference count to zero, it also calls the C++ "delete" statement on its own this pointer and deletes the servant.

#### 3.7.2.6.5 Return Value

None.

### 3.7.2.6.6 Example

```
if(servant != NULL)
servant->_remove_ref();
```

# 3.7.3 TP Interface

The TP interface supplies a set of service methods that can be invoked by application code. This is the *only* interface in the TP Framework that can safely be invoked by application code. All other interfaces have callback methods that are intended to be invoked only by system code.

The purpose of this interface is to provide high-level calls that application code can call, instead of calls to underlying APIs provided by the Portable Object Adapter (POA), the CORBA Naming Service, and the Oracle Tuxedo system. By using these calls, programmers can learn a simpler API and are spared the complexity of the underlying APIs. The TP interface implicitly uses two features of the Oracle Tuxedo software that extend the CORBA APIs:

- Factories and the FactoryFinder object
- Factory-based routing

For information about the FactoryFinder object, see the section FactoryFinder Interface. For more information about factory-based routing, see *Oracle® Tuxedo Application Configuration Guide*.

- Usage Notes
- TP::application\_responsibility()
- TP::bootstrap()
- TP::close\_xa\_rm()
- TP::create\_active\_object\_reference()
- TP::create\_object\_reference()
- TP::deactivateEnable()
- TP::get\_object\_id ()
- TP::get\_object\_reference()
- TP::open xa rm()
- TP::orb()
- TP::register\_factory()
- TP::unregister\_factory()
- TP::userlog()

# 3.7.3.1 Usage Notes

 During server application initialization, the application constructs the object reference for an application factory. It then invokes the register\_factory() method, passing in the factory's object reference together with a factory id field. On server release (shutdown), the application uses the unregister factory() method to unregister the factory.

- The TP class is a C++ native class.
- The TP.h file contains the declarations and definitions for the TP class.

C++ Declarations (in TP.h)

The C++ mapping is as follows:

```
class TP {
public:
static CORBA::Object_ptr create_object_reference(
const char* interfaceName,
const char* stroid,
CORBA::NVList ptr criteria);
static CORBA::Object ptr create active object reference(
const char* interfaceName,
const char* stroid,
Tobj Servant servant);
static CORBA::Object ptr get object reference();
static void register factory(
CORBA:: Object ptr factory or,
const char* factory id);
static void unregister factory(
CORBA:: Object ptr factory or,
const char* factory id);
static void deactivateEnable()
static void deactivateEnable(
const char* interfaceName,
const char* stroid,
Tobj Servant servant);
static CORBA::ORB ptr orb();
static Tobj Bootstrap* bootstrap();
static void open xa rm();
static void close xa rm();
static int userlog(char*, ...);
static char* get object id(CORBA::Object ptr obj);
static void application responsibility(
Tobj Servant servant);
```

# 3.7.3.2 TP::application\_responsibility()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values



# 3.7.3.2.1 Synopsis

Tells the TP Framework that the application is taking responsibility for the servant's lifetime.



Do not use this method in applications written using Oracle Tuxedo release 8.0 or later; instead, use the Tobj\_ServantBase:: add ref() method.

### 3.7.3.2.2 C++ Binding

static void application responsibility (Tobj Servant servant);

# **3.7.3.2.3** Arguments

#### servant

A pointer to a servant that is already known to the TP Framework.

### 3.7.3.2.4 Exceptions

#### TobjS::InvalidServant

Indicates that the specified servant is NULL.

# 3.7.3.2.5 Description

This method tells the TP Framework that the application is taking responsibility for the servant's lifetime. As a result of this call, when the TP Framework has completed deactivating the object (that is, after invoking the servant's deactivate\_object method), the TP Framework does nothing more with the object.

Once an application has taken responsibility for a servant, the application must take care to delete servant when it is no longer needed, the same as for any other C++ instance.

If the servant is not known to the TP Framework (that is, it is not active), this call has no effect.

#### 3.7.3.2.6 Return Values

None.

# 3.7.3.3 TP::bootstrap()

- Synopsis
- C++ Binding
- Arguments
- Return Value
- Exceptions
- Description



# 3.7.3.3.1 Synopsis

Returns a pointer to a Tobj::Tobj\_Bootstrap object. The Bootstrap object is used to access initial object references for the FactoryFinder object, the Interface Repository, the TransactionCurrent, and the SecurityCurrent objects.

### 3.7.3.3.2 C++ Binding

```
static Tobj Bootstrap* TP::bootstrap();
```

### 3.7.3.3.3 Arguments

None.

#### 3.7.3.3.4 Return Value

Upon successful completion, bootstrap() returns a pointer to the Tobj::Tobj\_Bootstrap object that is created by the TP Framework when the server application is started.

#### 3.7.3.3.5 Exceptions

None.

### 3.7.3.3.6 Description

The TP Framework creates a Tobj::Tobj\_Bootstrap object as part of initialization; it is not necessary for the application code to create any other Tobj::Tobj\_Bootstrap objects in the server.

Caution: Because the TP Framework owns the Tobj::Tobj\_Bootstrap object, server application code must not dispose of the Bootstrap object.



If you are using the CORBA INS bootstrap mechanism and you are not using the SecurityCurrent for security or TransactionCurrent for transactions, you do not need to use the Bootstrap object.

# 3.7.3.4 TP::close xa rm()

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- Exceptions

# 3.7.3.4.1 Synopsis

Closes the XA resource manager to which the invoking process is linked.

### 3.7.3.4.2 C++ Binding

static void TP::close xa rm ();

#### 3.7.3.4.3 Arguments

None.

#### 3.7.3.4.4 Description

The close\_xa\_rm() method closes the XA resource manager to which the invoking process is linked. XA resource managers are provided by database vendors, such as Oracle and Informix.

#### Note:

The functionality of this call is also provided by

Tobj::TransactionCurrent::close\_xa\_rm(). The TP::close\_xa\_rm() method provides a more convenient way for a server application to close a resource manager because there is no need to obtain an object reference to the TransactionCurrent object. A reference to the TransactionCurrent object can be obtained from the Bootstrap object. See TP::bootstrap() for an explanation of how to obtain a reference to the Bootstrap object. For more information about the TransactionCurrent object, see the CORBA Bootstrapping Programming Reference section and Using CORBA Transactions.

This method should be invoked once from the Server::release() method for each server that is involved in global transactions. This includes servers that are linked with an XA resource manager, as well as servers that are involved in global transactions, but are not actually linked with an XA-compliant resource manager.

The close\_xa\_rm() method should be invoked in place of a close invocation that is specific to the resource manager. Because resource managers differ in their initialization semantics, the specific information needed to close a particular resource manager is placed in the CLOSEINFO parameter in the GROUPS section of the Oracle Tuxedo system UBBCONFIG file.

The format of the CLOSEINFO string is dependent on the requirements of the database vendor providing the underlying resource manager. For more information about the CLOSEINFO parameter, see Setting Up a Oracle Tuxedo Application and the ubbconfig (5) reference page in the File Formats, Data Descriptions, MIBs, and System Processes Reference. Also, refer to database vendor documentation for information about how to develop and install applications that use the XA libraries.

#### 3.7.3.4.5 Return Values

None.

### 3.7.3.4.6 Exceptions

CORBA::BAD\_INV\_ORDER

There is an active transaction. The resource manager cannot be closed while a transaction is active.



#### Tobj::RMFailed

The tx close() call returned an error return code.



Unlike other exceptions returned by the TP Framework, the <code>Tobj::RMFailed</code> exception is defined in <code>tobj\_c.h</code> (which is derived from <code>tobj.idl</code>), not <code>TobjS\_c.h</code> (which is derived from <code>TobjS.idl</code>). This is because native clients can also open XA resource managers. Therefore, the exception returned is consistent with the exception expected by native client code and by <code>Server::release()</code> if it uses the alternate mechanism, <code>TransactionCurrent::close\_xa\_rm</code>, which is shared with native clients.

# 3.7.3.5 TP::create\_active\_object\_reference()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Caution
- Return Value

# 3.7.3.5.1 Synopsis

Creates an object reference and preactivates an object.

# 3.7.3.5.2 C++ Binding

```
static CORBA::Object_ptr
create_active_object_reference(
const char* interfaceName,
const char* stroid,
Tobj Servant servant);
```

# 3.7.3.5.3 Arguments

#### interfaceName

Specifies a character string that contains the fully qualified interface name for the object.

#### stroid

Specifies the <code>ObjectId</code> in string format. The <code>ObjectId</code> uniquely identifies this instance of the class. The programmer decides what information to place in the <code>ObjectId</code>. One possibility would be to use it to hold a database key. Choosing the value of an object identifier, and the degree of uniqueness, is part of the application design. The Oracle Tuxedo software cannot guarantee any uniqueness in object references, since these may be legitimately copied and shared outside the Oracle Tuxedo environment, for example by stringifying the object reference.

#### servant

A pointer to a servant that the application has already created and initialized.

### 3.7.3.5.4 Exceptions

#### TobjS::InvalidInterface

Indicates that the specified interfaceName is NULL.

#### TobjS::InvalidObjectId

Indicates the specified stroid is NULL.

#### TobjS::ServantAlreadyActive

The object could not be activated explicitly because the servant is already being used with another <code>ObjectId</code>. A servant can be used only with a single <code>ObjectId</code>. To preactivate objects containing different <code>ObjectIds</code>, the application must create multiple servants and preactivate them separately, one per <code>ObjectId</code>.

#### TobjS::ObjectAlreadyActive

The object could not be activated explicitly because the <code>ObjectId</code> is already being used in the Active Object Map. A given <code>ObjectId</code> can have only one servant associated with it. To change to a different servant, the application must first deactivate the object and activate it again.

#### TobjS::IllegalOperation

The object could not be activated explicitly because it does not have the process activation policy.

### 3.7.3.5.5 Description

This method creates an object reference and preactivates an object. The resulting object reference may be passed to clients who will use it to access the object.

Ordinarily, the application will call this method in two places:

- In Server::initialize() to preactivate process objects so that they do not need activation on the first invocation.
- In any method that creates object references to be returned to clients.

This method allows an application to activate an object explicitly before its first invocation. (For reasons you might want to do this, refer to the section <code>Explicit Activation.</code>) The user first creates a servant and sets its state before calling <code>create\_active\_object\_reference</code>. The TP Framework then enters the servant and string <code>ObjectId</code> in the Active Object Map. The result is exactly the same as if the TP Framework had previously invoked <code>Server::create\_servant</code>, received back the servant pointer, and then had invoked <code>servant::activate object</code>.

The object so activated must be for an interface that was declared with the process activation policy; otherwise, an exception is raised.

If the object is deactivated, an object reference held by a client might cause the object to be activated again in some other process. For a discussion about situations in which this might be a problem, refer to the section Explicit Activation.



#### Note:

There is one restriction on this method when the user-controlled concurrency policy option is set in the ICF file (See Parallel Objects.). The

TP::create\_active\_object\_reference method throws a TobjS::IllegalOperation exception if it is passed an interface with user-controlled concurrency set. Since the AOM is not used when user-controlled concurrency is set, there is no way for the TP Framework to connect an active object to this server.

#### 3.7.3.5.6 Caution

When you preactivate objects in an interface, you must specify an activation policy of process in the ICF file for that interface. However, when you specify the process activation policy for an interface in the ICF file, this can lead to the following problem.

- Problem Statement
- Workaround

#### 3.7.3.5.6.1 Problem Statement

- 1. You write SERVER1 such that all objects on interface A are preactivated. To prevent the object from being activated on demand by the TP Framework, you write the interface's activate\_object method to always throw the ActivateObjectFailed exception.
- 2. SERVER2 also implements objects of interface A. However, instead of preactivating the objects, SERVER2 lets the TP Framework activate them on demand.
- 3. If the administrator configures SERVER1 and SERVER2 in the same group, then a client can get an interface A object reference from SERVER2 and invoke on it. Then, due to load balancing, SERVER1 could be asked to activate an object on interface A. However, SERVER1 is not able to activate an object on interface A on demand because its activate object method throws the ActivateObjectFailed exception.

#### 3.7.3.5.6.2 Workaround

You can avoid this problem by having the administrator configure SERVER1 and SERVER2 in different groups. The administrator uses the SERVERS section of the UBBCONFIG file to define groups.

#### 3.7.3.5.7 Return Value

The newly created object reference.

# 3.7.3.6 TP::create\_object\_reference()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value
- Example



# 3.7.3.6.1 Synopsis

Creates an object reference. The resulting object reference may be passed to clients who use it to access the object.

### 3.7.3.6.2 C++ Binding

```
static CORBA::Object_ptr TP::create_object_reference (
const char* interfaceName,
const char* stroid,
CORBA::NVList ptr criteria);
```

### 3.7.3.6.3 Arguments

#### interfaceName

Specifies a character string that contains the fully qualified interface name for the object.

The interface name can be retrieved by making a call on the following interface typecode ID function: const\_char\* \_tc\_<CORBA interface name>::id(); where <CORBA interface name> is any object class name. For example:

```
char* idlname = tc Simple->id();
```

#### stroid

Specifies the <code>ObjectId</code> in string format. The <code>ObjectId</code> uniquely identifies this instance of the class. It is up to the programmer to decide what information to place in the <code>ObjectId</code>. One possibility would be to use the <code>ObjectId</code> to hold a database key. Choosing the value of an object identifier, and the degree of uniqueness, is part of the application design. The Oracle Tuxedo software cannot guarantee any uniqueness in object references, since object references may be legitimately copied and shared outside the Oracle Tuxedo domain (for example, by passing the object reference as a string). It is strongly recommended the you choose a unique <code>ObjectId</code> in order to allow parallel execution of invokes on object references.



The restriction on the length of the ObjectId has been removed in this release.

#### criteria

Specifies a list of named values that can be used to provide factory-based routing for the object reference. The list is optional and is of type CORBA::NVList. The use of factory-based routing is optional and is dependent on the use of this argument. If you do not want to use factory-based routing, you can pass a value of 0 (zero) for this argument.

The Oracle Tuxedo system administrator configures factory-based routing by specifying routing rules in the <code>UBBCONFIG</code> file. See Oracle® Tuxedo Application Configuration Guide online document for details on this facility.

# 3.7.3.6.4 Exceptions

The following exceptions can be raised by the create object reference() method:



#### InvalidInterface

Indicates that the specified interfaceName is NULL.

#### InvalidObjectId

Indicates that the specified stroid is NULL.

### 3.7.3.6.5 Description

The server application is responsible for invoking the <code>create\_object\_reference()</code> method. This method creates an object reference. The resulting object reference may be passed to clients who will use it to access the object.

Ordinarily, the server application calls this method in two places:

- In Server::initialize() to create factories for the server.
- In factory methods to create object references to be returned to clients.

For examples of how and when to call the <code>create\_object\_reference()</code> method, see Creating CORBA Server Applications.

#### 3.7.3.6.6 Return Value

#### Object

The newly created object reference.

# 3.7.3.6.7 Example

The following example shows how to use the criteria argument:

```
CORBA::NVList ptr criteria;
CORBA::Long branch id = 7;
CORBA::Long account id = 10001;
CORBA:: Any any val;
// Create the list and assign to var to cleanup on exit
CORBA::ORB::create list (2, criteria);
CORBA::NVList var criteria var(criteria);
// Add the BRANCH ID
any val <<= branch id;
criteria->add value("BRANCH ID", any val, 0);
// Add the ACCOUNT ID
any val <<= account id;
criteria->add value("ACCOUNT ID", any val, 0);
// Create the object reference.
TP::create object reference ("IDL:BankApp/Teller:1.0",
"Teller 01", criteria);
```

# 3.7.3.7 TP::deactivateEnable()

- Synopsis
- C++ Binding

- Arguments
- Exceptions
- Description
- Return Value

# 3.7.3.7.1 Synopsis

Enables application-controlled deactivation of CORBA objects.

### 3.7.3.7.2 C++ Binding

#### Current-object format:

```
static void TP::deactivateEnable();
```

#### Any-object format:

```
static void TP::deactivateEnable(
const char* interfaceName,
const char* stroid,
Tobj Servant servant);
```

### 3.7.3.7.3 Arguments

#### interfaceName

Specifies a character string that contains the fully qualified interface name for the object.

#### stroid

Specifies the ObjectId in string format for the object to be deactivated.

#### servant

A pointer to the servant associated with the stroid.

### 3.7.3.7.4 Exceptions

The following exceptions can be raised by the deactivateEnable() method:

#### IllegalOperation

Indicates that the TP::deactivateEnable method was invoked by an object with the activation policy set to transaction.

#### TobjS::ObjectNotActive

In the Any-object format, the object specified could not be deactivated because it was not active (the stroid and servant parameters did not identify an object that was in the Active Object Map).

# 3.7.3.7.5 Description

This method can be used to cause deactivation of an object, either the object currently executing (upon completion of the method in which it is called) or another object. It can only be used for objects with the process activation policy. It provides additional flexibility for objects with the process activation policy.



#### Note:

For single-threaded servers, the TP::deactivateEnable(interface, object id, servant) method can be used to deactivate an object. However, if that object is currently in a transaction, the object will be deactivated when the transaction commits or rolls back. If an invoke occurs on the object before the transaction is committed or rolled back, the object will not be deactivated. To ensure the desired behavior, make sure that the object is not in a transaction or ensure that no invokes occur on the object after the TP::deactivateEnable() call until the transaction is complete.

For multithreaded servers, use of the TP::deactivateEnable(interface, object id, servant) method is not supported for deactivation of objects in per-object servers. This method is supported for deactivation objects in per-request servers, however, the deactivation may be delayed because others threads are acting on the object.

Depending on which of the overloaded functions are called, the actions are as follows:

#### Current-object format

When called from within a method of an object with process activation policy, the object currently executing will be deactivated after completing the method being executed.

When called from within a method of an object with method activation, the effect is the same as the normal behavior of such objects (effectively, a NOOP).

When the object is deactivated, the TP Framework first removes the object from the Active Object Map. It then calls the associated servant's deactivate\_object method with a reason of DR METHOD END.

#### Any-object format

The application can request deactivation of an object by specifying its <code>ObjectId</code> and the associated servant.

If the object is currently executing, the TP Framework marks it for deactivation and waits until the object's method completes before deactivating the object (as with the "current-object format"). If the object is not currently executing, the TP Framework may deactivate it immediately. No indication is given to the caller as to the status of the deactivation. When the object is deactivated, the TP Framework first removes the object from the Active Object Map. It then calls the associated servant's <code>deactivate\_object</code> method with a reason of <code>DR\_EXPLICIT\_DEACTIVATE</code>.

If the object for which the deactivate is requested has a transaction activation policy, an IllegalOperation exception is raised. This is because deactivation of such objects may interfere with their correct notification of transaction completion by the Oracle Tuxedo transaction manager.

#### 3.7.3.7.6 Return Value

None.

# 3.7.3.8 TP::get object id ()

- Synopsis
- C++ Binding



- Arguments
- Exception
- Description
- Return Value

### 3.7.3.8.1 Synopsis

Allows a server to retrieve the string <code>ObjectId</code> contained in an object reference that was created in the TP Framework.

### 3.7.3.8.2 C++ Binding

```
char* TP::get_object_id(Corba::Object_ptr obj);
```

### 3.7.3.8.3 Arguments

obj

The object reference from which to get the ObjectId.

### 3.7.3.8.4 Exception

TobjS::InvalidObject

The object is nil or was not created by the TP Framework

### 3.7.3.8.5 Description

This method allows a server to retrieve the string <code>ObjectId</code> contained in an object reference that was created in the TP Framework. If the object reference was not created in the TP Framework (for example, it was created by a client ORB), an exception is raised.

The caller must call CORBA::string\_free on the returned value when the object reference is no longer needed.

#### 3.7.3.8.6 Return Value

The string ObjectId passed to TP::create\_object\_reference or TP::create active object reference when the object reference was created.

# 3.7.3.9 TP::get\_object\_reference()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 3.7.3.9.1 Synopsis

Returns a pointer to the current object.

### 3.7.3.9.2 C++ Binding

static CORBA::Object ptr TP::get object reference ();

### 3.7.3.9.3 Arguments

None.



If get\_object\_reference() is invoked from within either Server::initialize() or Server::release(), it is considered to be invoked outside the scope of an application's TP object execution; therefore, the TobjS::NilObject exception is raised.

# 3.7.3.9.4 Exceptions

The following exception can be raised by the get\_object\_reference() method:

#### NilObject

Indicates that the method was invoked outside the scope of an application's CORBA object execution. The reason string contains OutOfScope.

# 3.7.3.9.5 Description

This method returns a pointer to the current object. The CORBA::Object\_ptr pointer that is returned can be passed to a client.

#### 3.7.3.9.6 Return Value

The <code>get\_object\_reference()</code> method returns a <code>CORBA::Object\_ptr</code> for the current object when invoked within the scope of a CORBA object execution. Otherwise, the <code>TobjS::NilObject exception</code> is raised.

# 3.7.3.10 TP::open\_xa\_rm()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values

# 3.7.3.10.1 Synopsis

Opens the XA resource manager to which the invoking process is linked.

# 3.7.3.10.2 C++ Binding

static void TP::open xa rm();

### 3.7.3.10.3 Arguments

None.

#### 3.7.3.10.4 Exceptions

Tobj::RMFailed

The tx open() call returned an error return code.



Unlike other exceptions returned by the TP Framework, this exception is defined in tobj\_c.h (which is derived from tobj.idl), not in TobjS\_c.h (which is derived from TobjS.idl). This is because native clients can also open XA resource managers. Therefore, the exception returned is consistent with the exception expected by native client code and by Server::release() if it uses the alternate mechanism, TransactionCurrent::close xa rm, which is shared with native clients.

#### 3.7.3.10.5 Description

The <code>open\_xa\_rm()</code> method opens the XA resource manager to which the invoking process is linked. XA resource managers are provided by database vendors, such as Oracle and Informix.

#### Note:

The functionality of this method is also provided by

Tobj::TransactionCurrent::close\_xa\_rm(). However, TP::open\_xa\_rm() provides a more convenient way for a server application to close a resource manager because there is no need to obtain an object reference to the TransactionCurrent object. A reference to the TransactionCurrent object can be obtained from the Bootstrap object. See TP::bootstrap() for an explanation of how to obtain a reference to the Bootstrap object. For more information about the TransactionCurrent object, see the CORBA Bootstrapping Programming Reference section and Using CORBA Transactions.

This method should be invoked once from the <code>Server::initialize()</code> method for each server that participates in a global transaction. This includes servers that are linked with an XA resource manager, as well as servers that participate in a global transaction, but are not actually linked with an XA-compliant resource manager.

The <code>open\_xa\_rm()</code> method should be invoked in place of an open invocation that is specific to a resource manager. Because resource managers differ in their initialization semantics, the specific information needed to open a particular resource manager is placed in the <code>OPENINFO</code> parameter in the <code>GROUPS</code> section of the <code>UBBCONFIG</code> file.

The format of the OPENINFO string is dependent on the requirements of the database vendor providing the underlying resource manager. For more information about the CLOSEINFO parameter, see Oracle® Tuxedo Application Configuration Guide and the ubbconfig (5) reference page in the File Formats, Data Descriptions, MIBs, and System Processes



Reference. Also, refer to database vendor documentation for information about how to develop and install applications that use the XA libraries.



Only one resource manager can be linked to the invoking process.

### 3.7.3.10.6 Return Values

None.

# 3.7.3.11 TP::orb()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 3.7.3.11.1 Synopsis

Returns a pointer to an ORB object.

# 3.7.3.11.2 C++ Binding

static CORBA::ORB\_ptr TP::orb();

#### 3.7.3.11.3 Arguments

None.

# 3.7.3.11.4 Exceptions

None.

# 3.7.3.11.5 Description

Access to the <code>ORB</code> object allows the application to invoke ORB operations, such as  $string\_to\_object()$  and  $object\_to\_string()$ .



Because the TP Framework owns the ORB object, the application must not delete it.

#### 3.7.3.11.6 Return Value

Upon successful completion, orb() returns a pointer to the ORB object that is created by the TP Framework when the server program is started.

# 3.7.3.12 TP::register factory()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

### 3.7.3.12.1 Synopsis

Locates the Oracle Tuxedo FactoryFinder object and registers an Oracle Tuxedo factory.

# 3.7.3.12.2 C++ Binding

```
static void TP::register_factory(
CORBA::Object_ptr factory_or, const char* factory_id);
```

# 3.7.3.12.3 Arguments

#### factory or

Specifies the object reference that was created for an application factory using the TP::create object reference() method.

#### factory id

Specifies a string identifier that is used to identify the application factory. For some suggestions as to the composition of this string, see *Creating CORBA Server Applications*.

# 3.7.3.12.4 Exceptions

The following exceptions can be raised by the register factory () method:

#### TobjS::CannotProceed

Indicates that the FactoryFinder encountered an internal error during the search, with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the NameManager may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If the NameManager has terminated, and there is another NameManager running, start a new one. If no NameManagers are running, restart the application.

#### TobjS::InvalidName

Indicates that the  ${\tt id}$  string is empty. It is also raised if the field contains blank spaces or control characters.

#### TobjS::InvalidObject

Indicates that the factory value is nil.



#### TobjS::RegistrarNotAvailable

Indicates that the FactoryFinder object cannot locate the NameManager. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.



Another possible reason that this exception might occur is that the FactoryFinder cannot participate in a transaction. Therefore, you may need to suspend the current transaction before issuing the  $\mathtt{TP::register\_factory()}$  and

TP::unregister\_factory() calls. For information on suspending and resuming transactions, see *Using CORBA Transactions* in the online documentation.

#### TobjS::OverFlow

Indicates that the id string is longer than 128 bytes (currently the maximum allowable length).

### 3.7.3.12.5 Description

This method locates the Oracle Tuxedo FactoryFinder object and registers an Oracle Tuxedo factory. Typically, TP::register\_factory() is invoked from Server::initialize() when the server creates its factories. The register\_factory() method locates the Oracle Tuxedo FactoryFinder object and registers the Oracle Tuxedo factory.



#### **Caution:**

Callback objects (that is, those created by a joint client/server directly through the POA) should not be registered with a FactoryFinder.

#### 3.7.3.12.6 Return Value

None.

# 3.7.3.13 TP::unregister factory()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

### 3.7.3.13.1 Synopsis

Locates the Oracle Tuxedo FactoryFinder object and removes a factory.

### 3.7.3.13.2 C++ Binding

### 3.7.3.13.3 Arguments

#### factory or

Specifies the object reference that was created for an application factory using the TP::create object reference() method.

#### factory id

Specifies a string identifier that is used to identify the application factory. For some suggestions as to the composition of this string, see *Creating CORBA Server Applications*.

### 3.7.3.13.4 Exceptions

The following exceptions can be raised by the <code>unregister\_factory()</code> method:

#### CannotProceed

Indicates that the FactoryFinder encountered an internal error during the search, with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the NameManager may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If the NameManager has terminated, and there is another NameManager running, start a new one. If no NameManagers are running, restart the application.

#### InvalidName

Indicates that the id string is empty. It is also raised if the field contains blank spaces or control characters.

#### RegistrarNotAvailable

Indicates that the FactoryFinder object cannot locate the NameManager. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.



Another possible reason that this exception might occur is that the FactoryFinder cannot participate in a transaction. Therefore, you may need to suspend the current transaction before issuing the TP::register factory() and

TP::unregister\_factory() calls. For information on suspending and resuming transactions, see *Using CORBA Transactions* in the online documentation.

#### TobjS::OverFlow

Indicates that the id string is longer than 128 bytes (currently the maximum allowable length).

# 3.7.3.13.5 Description

This method locates the Oracle Tuxedo FactoryFinder object and removes a factory. Typically TP::unregister\_factory() is invoked from Server::release() to unregister server factories.

#### 3.7.3.13.6 Return Value

None.

# 3.7.3.14 TP::userlog()

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value
- Example

# 3.7.3.14.1 Synopsis

Writes a message to the user log (ULOG) file.

### 3.7.3.14.2 C++ Binding

```
static int TP::userlog(char*, ...);
```

# 3.7.3.14.3 Arguments

The first argument is a printf(3S) style format specification. The printf(3S) argument is described in a C or C++ reference manual.

# 3.7.3.14.4 Exceptions

None.

# 3.7.3.14.5 Description

The userlog() method writes a message to the user log (ULOG) file. Messages are appended to the ULOG file with a tag made up of the time (hhmmss), system name, process name, and process-id of the invoking process. The tag is terminated with a colon.

We recommend that server applications limit their use of userlog() messages to messages that can be used to help debug application errors; flooding the ulog file with incidental information can make it difficult to spot actual errors.

#### 3.7.3.14.6 Return Value

The userlog() method returns the number of characters that were output, or a negative value if an output error was encountered. Output errors include the inability to open or write to the current log file.



### 3.7.3.14.7 Example

The following example shows how to use the TP::userlog() method:

```
userlog ("System exception caught: %s", e.get id());
```

# 3.8 CosTransactions::TransactionalObject Interface Not Enforced

Use of this interface is now deprecated. Therefore, the use of this interface is now optional and no enforcement of descent from this interface is done for objects infected with transactions. The programmer can specify that an object is not to be infected by transactions by specifying the <code>never</code> or <code>ignore</code> transaction policies. There is no interface enforcement for eligibility for transactions. The only indicator is the transaction policy.



The CORBAservices Object Transaction Service does not require that all requests be performed within the scope of a transaction. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

# 3.9 Error Conditions, Exceptions, and Error Messages

- Exceptions Raised by the TP Framework
- Exceptions in the Server Application Code
- Exceptions and Transactions
- Restriction of Nested Calls on CORBA Objects

# 3.9.1 Exceptions Raised by the TP Framework

The following exceptions are raised by the TP Framework and are returned to clients when error conditions occur in, or are detected by, the TP Framework:

CORBA::INTERNAL

CORBA::OBJECT\_NOT\_EXIST

CORBA::OBJ\_ADAPTER

CORBA::INVALID\_TRANSACTION

CORBA::TRANSACTION ROLLEDBACK

Since the reason for these exceptions may be ambiguous, each time one of these exceptions is raised, the TP Framework also writes a descriptive error message that explains the reason to the user log file.

# 3.9.2 Exceptions in the Server Application Code

Exceptions raised within a method invoked by a client are always raised back to the client exactly as they were raised in the method invoked by the client.

The following TP Framework callback methods are initiated by events other than client requests on the object:

```
Tobj_ServantBase::activate_object()
Tobj_ServantBase::deactivate_object()
Server::create servant()
```

If exception conditions are raised in these methods, those exact exceptions are not reported back to the client. However, each of these methods is defined to raise an exception that includes a reason string. The TP Framework will catch the exception raised by the callback and log the reason string to the user log file. The TP Framework may raise an exception back to the client. Refer to the descriptions of the individual TP Framework callback methods for more information about these exceptions.

Example

# 3.9.2.1 Example

For Tobj\_ServantBase::deactivate\_object(), the following line of code throws a DeactivateObjectFailed exception:

```
throw TobjS::DeactivateObjectFailed( "deactivate failed to save state!");
```

This message is appended to the user log file with a tag made up of the time (hhmmss), system name, process name, and process-id of the calling process. The tag is terminated with a colon. The preceding throw statement causes the following line to appear in the user log file:

```
151104.T1!simpapps.247: APPEXC: deactivate failed to save state!
```

Where 151104 is the time (3:11:04pm), T1 is the system name, simpapps is the process name, 247 is the process-id, and APPEXC identifies the message as an application exception message.

# 3.9.3 Exceptions and Transactions

Exceptions that are raised in either CORBA object methods or in TP Framework callback methods will not automatically cause a transaction to be rolled back unless the TP Framework started the transaction. It is up to the application code to call  $Current.rollback\_only()$  if the condition that caused the exception to be raised should also cause the transaction to be rolled back.

# 3.9.4 Restriction of Nested Calls on CORBA Objects

The TP Framework restricts nested calls on CORBA objects. The restriction is as follows:

 During a client invocation of a method of CORBA object A, CORBA object A cannot be invoked by another CORBA object B that is acting as a client of CORBA object A.

The TP Framework will detect the fact that a second CORBA object is acting as a client to an object that is already processing a method invocation, and will return a CORBA::OBJ\_ADAPTER exception to the caller.

# Note:

Application code should not depend on this behavior; that is, users should not make any processing dependent on this behavior. This restriction may be lifted in a future release.



4

# CORBA Bootstrapping Programming Reference

#### ✓ Note:

The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- · for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

This topic includes the following sections:

- Why Bootstrapping Is Needed
- Supported Bootstrapping Mechanisms
- Oracle Bootstrapping Mechanism
- Bootstrap Object API
- Bootstrap Object Programming Examples
- Interoperable Naming Service Bootstrapping Mechanism

# 4.1 Why Bootstrapping Is Needed

To communicate with Oracle Tuxedo objects, a client application must obtain object references. Without an object reference, there can be no communication. To solve this problem, client applications use a bootstrapping mechanism to obtain object references to objects in an Oracle Tuxedo domain.

# 4.2 Supported Bootstrapping Mechanisms

In the Tuxedo 8.0 release and later, two bootstrapping mechanisms are supported:

- Oracle Bootstrapping Mechanism
   To use Oracle client ORB, you must use this mechanism.
- Interoperable Naming Service Bootstrapping Mechanism
   To use client ORB from another vendor, you must this mechanism.



The CORBA C++ client provided with Oracle Tuxedo software may use the Interoperable Naming Service bootstrapping mechanism, however, for performance reasons, this is not recommended.

# 4.3 Oracle Bootstrapping Mechanism

The Oracle bootstrapping mechanism uses the Bootstrap object. Bootstrap objects are local programming objects, not remote CORBA objects, in both the client and the server. When Bootstrap objects are created, their constructor requires the network address of an Oracle Tuxedo IIOP Listener/Handler. Given this information, the bootstrapping object can generate object references for the key remote objects in the Oracle Tuxedo domain. These object references can then be used to access services available in the Oracle Tuxedo domain.

- How Bootstrap Objects Work
- Types of Oracle Remote Clients Supported
- Capabilities and Limitations

# 4.3.1 How Bootstrap Objects Work

Bootstrap objects are created by a client or a server application that must access object references to the following Oracle Tuxedo CORBA interfaces:

- FactoryFinder
- Security
- Interface Repository
- Naming Service
- Notification Service
- Tobj SimpleEvents Service
- Transaction

Bootstrap objects may represent the first connection to a specific Oracle Tuxedo domain depending on the format of the IIOP Listener/Handler address. If the NULL scheme Universal Resource Locator (URL) format is used (the only address format supported in releases of Oracle WebLogic Enterprise prior to version 5.1 and Oracle Tuxedo release 8.0), the Bootstrap objects represent the first connection. However, if the URL format is used, the connection will not occur until after creation of the Bootstrap object. For more information on address formats and connection times, refer to Tobj Bootstrap.

For an Oracle Tuxedo CORBA remote client, Bootstrap objects are created with the host and the port for the Oracle Tuxedo IIOP Listener/Handler. However, for Oracle Tuxedo native client and server applications, there is no need to specify a host and port because they execute in a specific Oracle Tuxedo domain. The IIOP Listener/Handler host and the port ID are included in the Oracle Tuxedo domain configuration information.

After they are created, Bootstrap objects satisfy requests for object references for objects in a particular Oracle Tuxedo domain. Different Bootstrap objects allow the application to use multiple domains.

Using the Bootstrap object, you can obtain references to the following objects:

#### SecurityCurrent

The SecurityCurrent object is used to establish a security context within an Oracle Tuxedo domain. The client can then obtain the PrincipalAuthenticator from the principal authenticator attribute of the SecurityCurrent object.

#### TransactionCurrent

The TransactionCurrent object is used to participate in an Oracle Tuxedo transaction. The basic operations are as follows:

Begin

Begin a transaction. Future operations take place within the scope of this transaction.

Commit

End the transaction. All operations on this client application have completed successfully.

Roll back

Abort the transaction. Tell all other participants to roll back.

Suspend

Suspend participation in the current transaction. This operation returns an object that identifies the transaction and allows the client application to resume the transaction later.

Resume

Resume participation in the specified transaction.

#### FactoryFinder

The FactoryFinder object is used to obtain a factory. In the Oracle Tuxedo system, factories are used to create application objects. The FactoryFinder provides the following different methods to find factories:

- Get a list of all available factories that match a factory object reference (find factories)
- Get the factory that matches a name component consisting of id and kind (find one factory).
- Get the first available factory of a specific kind (find one factory by id).
- Get a list of all available factories of a specific kind (find factories by id).
- Get a list of all registered factories (list factories).

#### InterfaceRepository

The Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the Oracle Tuxedo domain. Clients using the Dynamic Invocation Interface (DII) need a reference to the Interface Repository to be able to build CORBA request structures.

#### NamingService

A NamingService object is used to obtain a reference to the root namespace. When you use this object, the ORB locates the root of the namespace.

#### NotificationService

The NotificationService object is used to obtain a reference to the event channel factory (CosNotifyChannelAdmin::EventChannelFactory) in the CosNotification Service. In the Oracle Tuxedo system, the EventChannelFactory is used to locate the Notification Service channel.

#### Tobi SimpleEventsService

The Tobj\_SimpleEventsService object is used to obtain a reference to the event channel factory (Tobj SimpleEvents::ChannelFactory) in the Oracle Simple Events Service. In the

Oracle Tuxedo system, the ChannelFactory is used to locate the Oracle Simple Events Service channel.

Using the bootstrapping mechanism, you can obtain six different references, as follows:

#### SecurityCurrent

The SecurityCurrent object is used to establish a security context within an Oracle Tuxedo domain. The client can then obtain the PrincipalAuthenticator from the principal authenticator attribute of the SecurityCurrent object.

#### TransactionCurrent

The TransactionCurrent object is used to participate in an Oracle Tuxedo transaction. The basic operations are as follows:

#### Begin

Begin a transaction. Future operations take place within the scope of this transaction.

#### Commit

End the transaction. All operations on this client application have completed successfully.

#### Roll back

Abort the transaction. Tell all other participants to roll back.

#### Suspend

Suspend participation in the current transaction. This operation returns an object that identifies the transaction and allows the client application to resume the transaction later.

#### Resume

Resume participation in the specified transaction.

#### FactoryFinder

The FactoryFinder object is used to obtain a factory. In Oracle Tuxedo CORBA, factories are used to create application objects. The FactoryFinder provides the following different methods to find factories:

- Get a list of all available factories that match a factory object reference (find factories).
- Get the factory that matches a name component consisting of id and kind (find\_one\_factory).
- Get the first available factory of a specific kind (find\_one\_factory\_by\_id).
- Get a list of all available factories of a specific kind (find factories by id).
- Get a list of all registered factories (list\_factories).

#### InterfaceRepository

The Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the Oracle Tuxedo domain. Clients using the Dynamic Invocation Interface (DII) need a reference to the Interface Repository to be able to build CORBA request structures. The ActiveX Client is a special case of this. Internally, the implementation of the COM/IIOP Bridge uses DII, so it must get the reference to the Interface Repository, although this is transparent to the desktop client.

#### NotificationService

The NotificationService object is used to obtain a reference to the event channel factory (CosNotifyChannelAdmin::EventChannelFactory) in the CosNotification Service. In Oracle Tuxedo CORBA, the EventChannelFactory is used to locate the Notification Service channel.

Tobj SimpleEventsService



The Tobj\_SimpleEventsService object is used to obtain a reference to the event channel factory (Tobj\_SimpleEvents::ChannelFactory) in the Oracle Simple Events Service. In Oracle Tuxedo CORBA, the ChannelFactory is used to locate the Oracle Simple Events Service channel.

The FactoryFinder and Interface Repository objects are not implemented in the environmental objects library. However, they are specific to an Oracle Tuxedo domain and are thus conceptually similar to the SecurityCurrent and TransactionCurrent objects in use.

The Bootstrap object implies an association or "session" between the client application and the Oracle Tuxedo domain. Within the context of this association, the Bootstrap object imposes a containment relationship with the other Current objects (or contained objects); that is, the SecurityCurrent and TransactionCurrent. Current objects are valid only for this domain and only while the Bootstrap object exists.

#### Note:

Resolving the SecurityCurrent when using the new URL address format (corbaloc://hostname:port\_number) is a local operation; that is, no connection is made by the client to the IIOP Listener/Handler.

In addition, a client can have only one instance of each of the Current objects at any time. If a Current object already exists, an attempt to create another Current object does not fail. Instead, another reference to the already existing object is handed out; that is, a client application may have more than one reference to the single instance of the Current object.

To create a new instance of a Current object, the application must first invoke the destroy\_current() method on the Bootstrap object. This invalidates all of the Current objects, but does not destroy the session with the Oracle Tuxedo domain. After invoking destroy\_current(), new instances of the Current objects can be created within the Oracle Tuxedo domain using the existing Bootstrap object.

To obtain Current objects for another domain, a different Bootstrap object must be constructed. Although it is possible to have multiple Bootstrap objects at one time, only one Bootstrap object may be "active;" that is, have Current objects associated with it. Thus, an application must first invoke <code>destroy\_current()</code> on the "active" Bootstrap object before obtaining new Current objects on another Bootstrap object, which then becomes the active Bootstrap object.

#### Note:

If you want to access objects in multiple domains, either import the object to the local domain or administratively configure your application access multiple domains. For more information on multi-domain configurations, see "Configuring Multiple CORBA Domains" in *Using the Oracle Tuxedo Domains Component*.

Servers and native clients are inside of the Oracle Tuxedo domain; therefore, no "session" is established. However, the same containment relationships are enforced. Servers and native clients access the domain they are currently in by specifying an empty string, rather than // host:port.





When using the Bootstrap object, client and server applications must use the Tobj\_Bootstrap::resolve\_initial\_references() method, not the ORB::resolve initial references() method.

# 4.3.2 Types of Oracle Remote Clients Supported

The following table shows the types of remote clients that can use the Bootstrap object to access the other environmental objects, such as FactoryFinder, SecurityCurrent, TransactionCurrent, and InterfaceRepository. These clients are provided with the Oracle Tuxedo CORBA software. Third-party client ORBs should use the CORBA Interoperable Naming Service.

Table 4-1 Oracle Remote Clients Supported

| Client    | Description  |
|-----------|--|
| CORBA C++ | CORBA C++ client applications use the Oracle Tuxedo C++ environmental objects to access the CORBA objects in an Oracle Tuxedo domain, and the Oracle Tuxedo Object Request Broker (ORB) to process from CORBA objects. Use the Oracle Tuxedo system development commands to build these client applications (see the Oracle Tuxedo Command Reference). |

# 4.3.3 Capabilities and Limitations

Bootstrap objects have the following capabilities and limitations:

- Multiple Bootstrap objects can coexist in a client application, although only one Bootstrap object can own the Current objects (Transaction and Security) at one time. Client applications must invoke destroy\_current() on the Bootstrap object associated with one domain before obtaining the Current objects on another domain. Although it is possible to have multiple Bootstrap objects that establish connections to different Oracle Tuxedo domains, only one set of Current objects is valid. Attempts to obtain other Current objects without destroying the existing Current objects fail.
- Method invocations to any Oracle Tuxedo domain that has security enabled other than the domain that provides the valid SecurityCurrent object will fail and return a CORBA::NO\_PERMISSION exception.
- Method invocations to any Oracle Tuxedo domain other than the domain that provides the valid TransactionCurrent object do not execute within the scope of a transaction.
- The transaction and security objects returned by the Bootstrap objects are Oracle
  implementations of the Current objects. If other ("native") Current objects are present in the
  environment, they are ignored.

# 4.4 Bootstrap Object API

The Bootstrap object application programming interface (API) is described first in terms of the OMG Interface Definition Language (IDL) (for portability), and then in C++. The C++ descriptions add the necessary constructor to build a Bootstrap object for a particular Oracle Tuxedo domain.

Tobj Module



- C++ Mapping
- Java Mapping
- Automation Mapping
- C++ Member Functions
- Java Methods
- Automation Methods

# 4.4.1 Tobj Module

The following table shows the object reference that is returned for each type ID.

Table 4-2 Returned Object References

| ID                           | Returned Object Reference for C++ Clients                                       |  |
|------------------------------|---|--|
| FactoryFinder                | FactoryFinder object (Tobj::FactoryFinder)                                      |  |
| InterfaceRepository          | InterfaceRepository object (CORBA::Repository)                                  |  |
| NameService                  | CORBA Naming Service (Tobj::NameService)  |  |
| NotificationService          | <pre>EventChannelFactory object (CosNotifyChannelAdmin::</pre>                  |  |
|                              | EventChannelFactory)  |  |
| SecuirtyCurrent              | SecurityCurrent object (SecurityLevel2::Current)                                |  |
| TransactionCurrent           | OTS Current object (Tobj::TransactionCurrent)                                   |  |
| Tobj_SimpleEvents<br>Service | Oracle Simple Events ChannelFactory object (Tobj_SimpleEvents:: ChannelFactory) |  |

The following table describes the Tobj module exceptions.

Table 4-3 Tobj Module Exceptions

| C++ Exception            | Java Exception                    | Description  |
|--------------------------|-----------------------------------|--|
| Tobj::<br>InvalidName    | com.beasys.Tobj.<br>InvalidName   | Raised if id is not one of the names specified in Table 4-2. On the server, resolve_initial_references also raises InvalidName when SecurityCurrent is passed. |
| Tobj::<br>InvalidDomain  | com.beasys.Tobj.<br>InvalidDomain | On the server application, raised if the Oracle Tuxedo server environment is not booted.   |
| CORBA::<br>NO_PERMISSION | org.omg.CORBA.<br>NO_PERMISSION   | Raised if id is TransactionCurrent or SecurityCurrent and another Bootstrap object in the client owns the Current objects.                                     |
| BAD_PARAM                | org.omg.CORBA.<br>BAD_PARAM       | Raised if the object is nil or if the hostname contained in the object does not match the connection.  |
| IMP_LIMIT                | org.omg.CORBA.<br>IMP_LIMIT       | Raised if the register_callback_port method is called more than once.  |



# 4.4.2 C++ Mapping

The following code snippet shows the C++ declarations in the Tobj bootstrap.h file.

```
#include <CORBA.h>
class Tobj_Bootstrap {
public:
Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
CORBA::Object_ptr resolve_initial_references(
const char* id);
void register_callback_port(CORBA::Object_ptr objref);
void destroy_current();
};
```

# 4.4.3 Java Mapping

The following code snippet shows the Tobj Bootstrap.java mapping.

```
package com.beasys;
public class Tobj Bootstrap {
public Tobj Bootstrap(org.omg.CORBA.ORB orb,
String address)
throws org.omg.CORBA.SystemException;
public class Tobj Bootstrap {
public Tobj Bootstrap(org.omg.CORBA.ORB orb, String address,
java.applet.Applet applet)
throws org.omg.CORBA.SystemException;
public void register callback port(orb.omg.CORBA.Object objref)
throws org.omg.CORBA.SystemException;
public org.omg.CORBA.Object
resolve initial references (String id)
throws Tobj. InvalidName,
org.omg.CORBA.SystemException;
public void destroy current()
throws org.omg.CORBA.SystemException;
```

# 4.4.4 Automation Mapping

The following code snippet shows Automation Bootstrap interface mapping.

```
interface DITobj_Bootstrap : IDispatch
{
HRESULT Initialize(
[in] BSTR address);
HRESULT CreateObject(
[in] BSTR progid,
[out, retval] IDispatch** rtrn);
HRESULT destroy_current();
};
```



## 4.4.5 C++ Member Functions

This section describes the C++ member functions supported by the Oracle bootstrapping mechanism.

- Tobj\_Bootstrap
- Tobj Bootstrap::register\_callback\_port
- Tobj Bootstrap::resolve initial references
- Tobj\_Bootstrap::destroy\_current()

# 4.4.5.1 Tobj Bootstrap

- Synopsis
- C++ Mapping
- Parameters
- Exception
- Description
- Return Values

## 4.4.5.1.1 Synopsis

The Bootstrap object constructor.

# 4.4.5.1.2 C++ Mapping

```
Tobj_Bootstrap(CORBA::ORB_ptr orb, const char* address);
throws Tobj::BAD_PARAM
org.omg.CORBA.SystemException;
```

#### 4.4.5.1.3 Parameters

#### orb

A pointer to the ORB object in the client. The Bootstrap object uses the <code>string\_to\_object</code> method of orb internally.

#### address

The address of the Oracle Tuxedo domain IIOP Listener/Handler.



Multiple Tobj Bootstraps going to the same domain is not supported.

The address is specified differently depending on the type of client and the level of security required. There can be three types of clients, as follows:



#### Remote client

For a description of the remote clients supported by Oracle Tuxedo CORBA, see the section Types of Oracle Remote Clients Supported.

For remote clients, address specifies the network address of an IIOP Listener/Handler through which client applications gain access to an Oracle Tuxedo domain.

The address may be specified in either of the following formats:

```
"//hostname:port_number"
"//#.#.#.#:port_number"
"corbaloc://hostname:port_number"
"corbalocs://hostname:port_number"
```

In the first format, the domain finds an address for *hostname* using the local name resolution facilities (usually DNS). The hostname must be the remote machine, and the local name resolution facilities must unambiguously resolve *hostname* to the address of the remote machine.



The hostname must begin with a letter character.

In the second format, the #.#.# is in dotted decimal format. In dotted decimal format, each # should be a number from 0 to 255. This dotted decimal number represents the IP address of the remote machine.

In both the first and second formats, *port\_number* is the TCP port number at which the domain process listens for incoming requests. The *port\_number* should be a number between 0 and 65535.

You can specify one or more TCP/IP addresses. You specify multiple addresses using a comma-separated list. For example:

```
//m1.acme:3050
//m1.acme:3050,//m2.acme:3050,//m3.acme:3051
```

If you specify multiple addresses, the Oracle Tuxedo software tries the addresses in order, left to right, until a connection is established. If a syntax error is detected in any of the addresses as it is being tried, a BAD\_PARAM exception is returned to the caller immediately and the Oracle Tuxedo software aborts the attempt to make a connection. For example, if the first address in the comma-separated list shown above were //m1.3050, a syntax error would be detected and the attempt to make a connection would be aborted. If the Oracle Tuxedo software encounters the end of the address list before it tries an address that is valid, that is, a connection cannot be made to any of the addresses listed, the INVALID\_DOMAIN exception is returned to the caller. If an exception other than INVALID\_DOMAIN is raised, it is returned to the caller immediately.

Oracle Tuxedo also supports random address selection. To use random address selection, you can specify any member of an address list as a grouping of pipe-separated () network addresses enclosed in parentheses. For example:

```
(//m1.acme:3050|//m2.acme:3050),//m1.acme:7000
```



When you use this format, the Oracle Tuxedo system randomly selects one of the addresses enclosed in parentheses, either <code>//ml.acme:3050</code> or <code>//m2.acme:3050</code>. If an exception other than <code>INVALID\_DOMAIN</code> is raised, it is returned to the caller immediately. If a connection cannot be made to the address selected, the next element that follows the addresses enclosed in parentheses is attempted. If the end of the string is encountered before a connection can be made, the <code>INVALID\_DOMAIN</code> exception is thrown to the caller.

#### Note:

If you specify an address list in the following format: (//m1.acme:3050||//m2.acme:3050), //r1.acme:7000 the NULL address in the pipe-separated list is considered invalid. If the Oracle Tuxedo software randomly selects the invalid address, the BAD\_PARAM exception is returned to the caller and the Oracle Tuxedo software aborts the connection attempt.

The address string can be specified either in the TOBJADDR environment variable or in the address parameter of the Tobj\_Bootstrap constructor.

For information about the TOBJADDR environment variable, see the section Managing Remote Client Applications in the Oracle Tuxedo Application Configuration Guide . However, the address specified in Tobj\_Bootstrap always take precedence over the TOBJADDR environment variable. To use the TOBJADDR environment variable to specify an address string, you must specify an empty string in the Tobj\_Bootstrap address parameter.

#### Note:

For C++ applications, TOBJADDR is an environment variable; for Java applications, it is a property; for Java applets, it is an HTML parameter.

The third and fourth formats are called Uniform Resource Locator (URL) address formats and were introduced in the Oracle WebLogic Enterprise version 5.1 release. As with the NULL scheme URL address format (//hostname:port\_number), you use the URL address formats to specify the location of the IIOP Listener/Handler. However, when the corbaloc URL address format is used, the client application's initial connection to the IIOP Listener/Handler is deferred until authentication of the principal's, or client's, identity or the first user initiated operation. Using the corbalocs URL address format has the same effect on the deferred connection time as corbaloc, but, additionally, the client application makes its initial connection to the ISL/ISH using the Secure Sockets Layer (SSL) protocol. The following table highlights the differences between the two URL address formats.



| URL Address<br>Formats | Differences in Mode of Operation   |  |
|------------------------|--|--|
| corbaloc               | Invocations to the IIOP Listener/Handler are unprotected. Configuring the IIOP Listener/Handler for the SSL protocol is optional.  |  |
|                        | Note:  A principal can secure the bootstrapping process by using the  SecurityLevel2::Current::authenticate()  operation to specify that certificate-based authentication is to be used. |  |
| corbalocs              | Invocations to the IIOP Listener/Handler are protected and the IIOP Listener/Handler or the server ORB must be configured to enable the use of the SSL protocol.                         |  |

These URL address formats are a subset of the definition of object URLs adopted by the OMG as part of the Interoperable Naming Service submission. The Oracle Tuxedo software also extends the URL format described in the OMG Interoperable Naming Service submission to support a secure form that is modeled after the URL for secure HTTP, as well as to support the randomize functionality that was added in the Oracle WebLogic Enterprise version 4.2.

The <code>corbaloc</code> and <code>corbalocs</code> URL schemes provide locations that are easily manipulated in both TCP/IP and DNS centric environments. These URL schemes contain a DNS-style hostname or IP address and a  $port_number$ . The following are some examples of the URL formats:

```
corbaloc://curly:1024,larry:1022,joe:1999
corbalocs://host1:1024,{host2:1022|host3:1999}
```

As an enhancement to the URL syntax described in the OMG Interoperable Naming Service submission, the Oracle WebLogic Enterprise version 5.1 software extended the syntax to support a list of multiple URLs, each with a different scheme. The following are some examples of the extension:

```
corbalocs://curly:1024,corbaloc://larry:1111,
corbalocs://ctxobj:3434,mthd:3434,corbaloc://force:1111
```

In the above example, if the parser reaches the URL <code>corbaloc://force:1111</code>, it resets its internal state as if it had never attempted secure connections and then begins attempting unprotected connections.

#### Caution:

Do not mix the use of NULL scheme URL addresses (//hostname:port number) with corbaloc and corbalocs URL addresses.

#### Note:

The Bootstrap object supplied for use with the Netscape embedded Java ORB and JavaSoft JDK ORB does not support corbaloc and corbalocs URLs.

For more information on using the corbaloc and corbalocs URL address formats, see Using Security in CORBA Applications.

The network address that is specified in the Bootstrap constructor or in TOBJADDR must exactly match the network address in the server application's UBBCONFIG file, both the address as well as the capitalization. If the addresses do not match, the invocation to the Bootstrap constructor will fail with the following seemingly unrelated error message:

ERROR: Unofficial connection from client at <tcp/ip address>/<port-number>

For example, if the network address is specified (using the NULL URL address format) as //TRIXIE: 3500 in the ISL command-line option string in the server application's UBBCONFIG file, specifying either //192.12.4.6:3500 or //trixie:3500 in the Bootstrap constructor or in TOBJADDR will cause the connection attempt to fail. On UNIX systems, use the uname -n command on the host system to determine the capitalization used. On Windows systems, see the host system's network settings in the Control Panel to determine the correct capitalization.

#### Note:

The network address that is specified in the Bootstrap constructor or in TOBJADDR must exactly match the network address in the server application's UBBCONFIG file, both the address as well as the capitalization. If the addresses do not match. the invocation to the Bootstrap constructor will fail with the following seemingly unrelated error message:

ERROR: Unofficial connection from client at <tcp/ip address>/<port-number>

For example, if the network address is specified (using the NULL URL address format) as //TRIXIE: 3500 in the ISL command-line option string in the server application's UBBCONFIG file, specifying either //192.12.4.6:3500 or //trixie:3500 in the Bootstrap constructor or in TOBJADDR will cause the connection attempt to fail. On UNIX systems, use the uname -n command on the host system to determine the capitalization used. On Windows systems, see the host system's network settings in the Control Panel to determine the correct capitalization.



The error in the previous note is deferred when the URL address format is used, that is, the error does not occur at the time of Bootstrap object construction because the connection to the ISL/ISH is deferred until later.

#### Native client

For a native client, the <code>address</code> parameter in the <code>Tobj\_Bootstrap</code> constructor must always be an empty string (not a NULL pointer). The native client connects to the application that is specified in the <code>TUXCONFIG</code> environment variable. The constructor raises <code>CORBA::BAD PARAM</code> if the address is not empty.

#### Server acting as a client

When servers need access to the Bootstrap object, they should obtain a reference to it using the TP framework by invoking  $\mathtt{TP.bootstrap}()$ . Servers should not attempt to create a new instance of the Bootstrap object.

#### applet (Applies to Java method only)

This is a pointer to the client applet. If the client applet does not explicitly pass the ISH host and port to the Bootstrap constructor, you can pass this argument, which causes the Bootstrap object to search for the TOBJADDR definition in the HTML file for the applet.

## 4.4.5.1.4 Exception

#### BAD PARAM

Raised if the object is nil or if the host contained in the object does not match the connection or the host address (//hostname:port number) is not in a valid format.

## 4.4.5.1.5 Description

A C++ member function (or Java method) that creates Bootstrap objects.

#### 4.4.5.1.6 Return Values

A pointer to a newly created Bootstrap object.

# 4.4.5.2 Tobj\_Bootstrap::register\_callback\_port

- Synopsis
- C++ Mapping
- Parameter
- Exceptions
- Description
- Usage Notes
- Return Values



## 4.4.5.2.1 Synopsis

Registers the joint client/server's listening port in IIOP Handler (ISH).

## 4.4.5.2.2 C++ Mapping

void register callback port(CORBA::Object ptr objref);

#### 4.4.5.2.3 Parameter

#### objref

The object reference created by the joint client/server.

### 4.4.5.2.4 Exceptions

#### BAD PARAM

Raised if the object is nil or if the host contained in the object does not match the connection.

#### IMP LIMIT

Raised if the register callback port method is called more than once.

### 4.4.5.2.5 Description

This C++ member function (or Java method) is called to notify the ISH of a listening port in the joint client/server. This method should only be used for joint client/server ORBs that do not support GIOP 1.2 bidirectional capabilities (that is GIOP 1.0 and 1.1 client ORBs). For GIOP 1.0 and 1.1, the ISH supports only one listening port per joint client/server; therefore, the register\_callback\_port method should only be called once per connected joint client/server.

## 4.4.5.2.6 Usage Notes

The following information must be given consideration when using this method:

- If the register\_callback\_port method is not invoked by the joint client/server, the callback port is not registered with the ISH and the server defaults to Asymmetric Outbound IIOP. In this case, you *must* start the server's IIOP Listener (ISL) with the -0 option. The -0 option enables Asymmetric outbound IIOP; otherwise, server-to-client invocations will not be allowed by the ISL/ISH.
- If you are using the OracleWrapper Callbacks API instead of the POA and you want to use bidirectional behavior, you always need to invoke the register\_callback\_port method, even when you are using a ISH that supports GIOP 1.2.
- If you want to use bidirectional capability for a callback object, you must invoke the register\_callback\_port method before you pass the callback object reference to the server.

#### 4.4.5.2.7 Return Values

None.

## 4.4.5.3 Tobj Bootstrap::resolve initial references

- Synopsis
- C++ Mapping



- Parameter
- Exceptions
- Description
- Return Values

### 4.4.5.3.1 Synopsis

Acquires CORBA object references.

### 4.4.5.3.2 C++ Mapping

```
CORBA::Object_ptr resolve_initial_references(
const char* id);
throws Tobj::InvalidName,
org.omg.CORBA.SystemException;
```

#### 4.4.5.3.3 Parameter

#### id

This parameter must be one of the following:

```
"FactoryFinder"
"InterfaceRepository"
"NameService"
"NotificationService"
"SecurityCurrent"
"TransactionCurrent"
"Tobj SimpleEventsService"
```

## 4.4.5.3.4 Exceptions

#### InvalidName

Raised if id is not one of the names specified above. On the server, resolve\_initial\_references also raises Tobj::InvalidName when SecurityCurrent is passed.

#### CORBA:: NO PERMISSION

Raised if id is TransactionCurrent or SecurityCurrent and another Bootstrap object in the client owns the Current objects.

## 4.4.5.3.5 Description

This C++ member function (or Java method) acquires CORBA object references for the FactoryFinder, SecurityCurrent, TransactionCurrent, NotificationService, Tobj\_SimpleEventsService, and InterfaceRepository objects. For the specific object reference, invoke the \_narrow function. For example, for FactoryFinder, invoke

Tobj::FactoryFinder:: narrow.

### 4.4.5.3.6 Return Values

The following table shows the object reference that is returned for each type id.

# 4.4.5.4 Tobj Bootstrap::destroy current()

- Synopsis
- C++ Mapping
- Exception
- Description
- Return Values

## 4.4.5.4.1 Synopsis

Destroys the Current objects for the domain represented by the Bootstrap object.

### 4.4.5.4.2 C++ Mapping

void destroy current();

## 4.4.5.4.3 Exception

Raises CORBA:: NO PERMISSION if the Bootstrap object is not the owner of the Current objects.

### 4.4.5.4.4 Description

This C++ member function invalidates the Current objects for the domain represented by the Bootstrap object. After invoking the <code>destroy\_current()</code> method, the Current objects are marked as invalid. Any subsequent attempt to use the old Current objects will throw the exception <code>CORBA::BAD\_INV\_ORDER</code>. Good programming practice is to release all Current objects before invoking <code>destroy\_current()</code>.



The <code>destroy\_current()</code> method must be invoked on the Bootstrap object for the domain that currently owns the two Current objects (Transaction and Security). This also results in an implicit invocation to <code>logoff</code> for security and implicitly rolls back any transaction that was begun by the client.

The application must invoke <code>destroy\_current()</code> before invoking <code>resolve\_initial\_references</code> for TransactionCurrent or SecurityCurrent on another domain; otherwise, <code>resolve\_initial\_references</code> raises <code>CORBA::NO\_PERMISSION</code>.

## 4.4.5.4.5 Return Values

None.

## 4.4.6 Java Methods

The Java Oracle bootstrapping API supports the following methods:

- Tobj Bootstrap
- Tobj Bootstrap.register callback port



- Tobj Bootstrap.resolve initial references
- Tobj\_Bootstrap.destroy\_current
- Tobj\_Bootstrap.GetTransactions
- Tobj\_Bootstrap.getUserTransaction
- Tobj Bootstrap.getNativeProperties
- Tobj\_Bootstrap.getRemoteProperties

## 4.4.7 Automation Methods

This section describes the Automation methods supported by the Oracle bootstrapping mechanism.

- Initialize
- CreateObject
- DestroyCurrent

### 4.4.7.1 Initialize

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameter
- Return Values
- Exceptions

## 4.4.7.1.1 Synopsis

Initializes the Bootstrap object into an Oracle Tuxedo domain.

# 4.4.7.1.2 MIDL Mapping

```
HRESULT Initialize(
[in] BSTR host);
```

## 4.4.7.1.3 Automation Mapping

Sub Initialize (address As String)

#### 4.4.7.1.4 Parameter

#### address

The host name and port of the Oracle Tuxedo domain IIOP Listener/Handler. One or more TCP/IP addresses can be specified. Multiple addresses are specified using a commaseparated list, as in the C++ mappings. If no address is specified, the value of the TOBJADDR environmental variable is used.



## Note:

The network address that is specified in the Bootstrap constructor or in TOBJADDR must exactly match the network address in the application's UBBCONFIG file, both the format of the address as well as the capitalization. If the addresses do not match, the invocation to the Bootstrap constructor will fail with the following seemingly unrelated error message:

ERROR: Unofficial connection from client at
<tcp/ip address>/<port-number>

For example, if the network address is specified as //TRIXIE:3500 in the ISL command-line option string, specifying either //192.12.4.6:3500 or //trixie:3500 in the Bootstrap constructor or in TOBJADDR will cause the connection attempt to fail. On UNIX systems, use the uname -n command on the host system to determine the capitalization used. On Windows systems, see the host system's network settings in the Control Panel to determine the correct capitalization.

#### 4.4.7.1.5 Return Values

None.

## 4.4.7.1.6 Exceptions

The following table describes the exceptions.

**Table 4-5 Initialize Exceptions** 

| HRESULT                    | Description                     | Meaning   |
|----------------------------|---------------------------------|---|
| ITF_E_NO_PERMIS SION_ YES  | Bootstrap already initialized   | The Bootstrap object has already been initialized. To connect to a new Oracle Tuxedo domain, you must create a new Bootstrap object.    |
| E_INVALIDARG               | Invalid address parameter       | The address supplied is not valid.  |
| E_OUTOFMEMOY               | Memory allocation failed        | The required memory could not be allocated.   |
| E_FAIL                     | Invalid domain                  | Unable to communicate with the Oracle Tuxedo domain at the address specified or TOBJADDR is not defined.                                |
| <system error=""></system> | Unable to obtain initial object | Unable to initialize the Bootstrap object. The system error causing the failure is returned in the "Number" member of the error object. |

# 4.4.7.2 CreateObject

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameter

- Return Value
- Exceptions

## 4.4.7.2.1 Synopsis

Creates an instance of a Current environmental object.

## 4.4.7.2.2 MIDL Mapping

```
HRESULT CreateObject(
[in] BSTR progid,
[out, retval] IDispatch** rtrn);
```

## 4.4.7.2.3 Automation Mapping

Function CreateObject(progid As String) As Object

#### 4.4.7.2.4 Parameter

#### progid

The progid of the environmental object to create. Valid progids are:

```
Tobj.FactoryFinder
Tobj.SecurityCurrent
Tobj.TransactionCurrent
```

### 4.4.7.2.5 Return Value

A reference to the interface pointer of the created environmental object.

## 4.4.7.2.6 Exceptions

The following table describes the exceptions.

Table 4-6 CreateObject Exceptions

| Exception                      | Description                          | Meaning   |
|--------------------------------|--------------------------------------|---|
| ITF_E_NO_PERMIS SION _YES      | Bootstrap object must be initialized | The Bootstrap object has not been initialized.  |
| ITF_E_NO_PERMIS<br>SION<br>_NO | No permission                        | If the progid specifies a transaction or security current and another Bootstrap object in the client owns the current objects.                      |
| E_INVALIDARG                   | Invalid<br>progidparameter           | The progid specified is not valid.  |
| E_INVALIDARG                   | Invalid name                         | The requested progid is not one of the valid parameter values specified above.  |
| E_INVALIDARG                   | Unknown object                       | The requested progid is not registered on your system.  |
| <system error=""></system>     | CoCreate<br>Instance()<br>failed     | The Bootstrap object could not create an instance of the requested object. The system error is returned in the "Number" member of the error object. |

## 4.4.7.3 DestroyCurrent

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameters
- Return Value
- Exceptions

## 4.4.7.3.1 Synopsis

Logs out of the Oracle Tuxedo domain and invalidates the TransactionCurrent and SecurityCurrent objects.

### 4.4.7.3.2 MIDL Mapping

```
HRESULT destroy current();
```

### 4.4.7.3.3 Automation Mapping

```
Sub destroy current()
```

#### 4.4.7.3.4 Parameters

None.

#### 4.4.7.3.5 Return Value

None.

### 4.4.7.3.6 Exceptions

None.

# 4.5 Bootstrap Object Programming Examples

This section provides the following programming examples that use Bootstrap objects.

Visual Basic Client Example: Using the Bootstrap Object

# 4.5.1 Visual Basic Client Example: Using the Bootstrap Object

The following code snippet shows how to program a Visual Basic client to use the Bootstrap object.

```
'Declare the Bootstrap object
Public oBootstrap As DITobj Bootstrap
```

'Declare the FactoryFinder object Public oBsFactoryFinder As DITobj\_FactoryFinder

```
'Declare factory for Registrar object
Public oRegistrarFactory As DIUniversityB RegistrarFactory
'Declare actual Registrar object
Public oRegistrarFactory As DIUniversityB RegistrarFactory
'Create the Bootstrap object
Set oBootstrap = CreateObject("Tobj.Bootstrap")
'Connect to the Oracle Tuxedo Domain
oBootstrap.Initialize "//host:port"
'Get the FactoryFinder for the Oracle Tuxedo Domain
Set oBSFactoryFinder = oBootstrap.CreateObject("Tobj.FactoryFinder")
'Get a factory for the Registrar object
'using the FactoryFinder method find one factory by id
Set oRegistrarFactory =
oBSFactoryFinder.find_one_factory_by_id("RegistrarFactoryID")
'Create a Registrar object
Set oRegistrar = oRegistrarFactory.find registrar(exc)
```

# 4.6 Interoperable Naming Service Bootstrapping Mechanism

This topic includes the following topics:

- Introduction
- INS Object References
- INS Command-line Options
- INS Initialization Operations
- INS Object URL Schemes
- Getting a FactoryFinder Object Reference Using INS
- Getting a PrincipalAuthenticator Object Reference Using INS
- Getting a TransactionFactory Object Reference Using INS

# 4.6.1 Introduction

As of release 8.0, the Oracle Tuxedo ORB supports the CORBA Naming Service bootstrapping mechanism (referred to in this document as the Interoperable Naming Service), as specified in Chapters 4 and 13 of the CORBA Specification revision 2.4.2.

This support enables ORBs that implement the Interoperable Naming Service (INS) bootstrapping mechanism to query the Oracle Tuxedo server-side ORB to get object references to initial objects such as FactoryFinder and to PrincipalAuthenticator to the Oracle Tuxedo environment. This support along with client support for interoperable initial object references enables clients to use the INS bootstrapping mechanism instead of the Oracle bootstrapping mechanism.





The CORBA C++ client provided with Oracle Tuxedo software may use the INS bootstrapping mechanism, however, for performance reasons, this is not recommended.

# 4.6.2 INS Object References

The following table shows the object reference that is returned for each type ID.

Table 4-7 Returned Object References

| ID                           | Returned Object Reference  |  |
|------------------------------|--|--|
| FactoryFinder                | FactoryFinder object (CORBA::FactoryFinder)  |  |
| InterfaceRepository          | <pre>InterfaceRepository object (CORBA::Repository)</pre>                          |  |
| NameService                  | CORBA Naming Service object (CORBA::NameService)                                   |  |
| NotificationService          | <pre>EventChannelFactory object (CosNotifyChannelAdmin::EventChannelFactory)</pre> |  |
| POACurrent                   | POACurrent object (CORBA::POACurrent)  |  |
| PrincipalAuthentica tor      | PrincipalAuthenticator object (SecurityLevel2::PrincipalAuthenticator)             |  |
| RootPOA                      | RootPOA object (CORBA::RootPOA)  |  |
| Tobj_SimpleEvents<br>Service | Oracle Simple Events ChannelFactory object (Tobj_SimpleEvents::ChannelFactory)     |  |

# 4.6.3 INS Command-line Options

As of release 8.0, Oracle Tuxedo CORBA supports the -ORBInitRef and -ORBDefaultInitRef command-line options. For a complete description of these options, see "ORB Initialization Member Function".

The following example assumes an Oracle Tuxedo CORBA IIOP client is talking to an Oracle Tuxedo CORBA IIOP server environment:

```
client_app -ORBid BEA_IIOP -ORBInitRef
FactoryFinder=corbaloc::myhost:2468/FactoryFinder
```

Given this example, a call to ORB::resolve\_initial\_references for the FactoryFinder will result in an interoperable initial reference request being sent to the ISL/ISH on myhost at port 2468.



The case of myhost must exactly match the case of the host specified for the ISL/ISH in the tuxconfig file.

# 4.6.4 INS Initialization Operations

To use the INS bootstrapping mechanism, applications programmers must observe the following requirements:

• Oracle Tuxedo CORBA IIOP clients that want to use the INS initial reference mechanism must now call ORB::resolve\_initial\_references function, instead of the Tobj\_Bootstrap::resolve\_initial\_references function. For a syntactical description of ORB::resolve\_initial\_references, see "CORBA::ORB::resolve\_initial\_references".

## Note:

The Tobj Bootstrap API is still supported and its behavior has not changed.

Oracle Tuxedo CORBA IIOP clients using the INS initial reference mechanism should use
the ORB::list\_initial\_services function instead of the
Tobj\_Bootstrap::list\_initial\_services function. For a syntactical description of
ORB::list\_initial\_services, see "CORBA::ORB::list\_initial\_services".

# 4.6.5 INS Object URL Schemes

As of release 8.0, Oracle Tuxedo CORBA supports an additional Uniform Resource Locator (URL) format to be used for the specification of the location for access to an Oracle Tuxedo CORBA server environment and from where to retrieve references to initial object. The new URL format both follows and extends the definition of object URLs adopted by the OMG as part of the INS specification. The URL format described in the INS specification has also been extended to support a secure form modeled after the URL for secure HTTP, as well as the ability to support the randomize functionality initially provided in Oracle WebLogic Enterprise version 5.1.

The CORBA 2.4.2 specification requires that three object URL schemes must be supported by a compliant ORB. These schemes are defined as IOR, corbaloc, and corbaname.



The new URL string formats may also be passed to the <code>ORB::string\_to\_object</code> function.

- IOR URL Scheme
- corbaloc URL Scheme
- corbaname URL Scheme

## 4.6.5.1 IOR URL Scheme

The IOR scheme takes the form of a string that is formatted as <code>IOR:hex\_octets</code>. The scheme name is IOR and the text after the ':' is defined in the CORBA specification. The IOR URL scheme is robust and insulates the client from the encapsulated transport information and object key used to reference the object.



## 4.6.5.2 corbaloc URL Scheme

It is difficult for humans to exchange IORs through nonelectronic means because of their lengths and the text encoding of binary information. The corbaloc and corbalocs URL schemes provide stringified object references in a format that is familiar to people and similar to the popular URL schemes of FTP and HTTP. The URL schemes defined for corbaloc and corbalocs are easily manipulated in both TCP/IP and DNS centric environments. The corbaloc and corbalocs URL contains:

- DNS-style host name or IP address and port
- The version of the IIOP protocol to be used (optional)
- An object key (optional)

By default, corbaloc URLs denote objects that can be contacted over IIOP, while corbalocs URLs denote objects that can be contacted using IIOP over SSL.

The following table lists the BNF syntax for each URLs element.

Table 4-8 BNF Format for URL Elements

| URL Element                                       | BNF Format  |
|---|---|
| <corbaloc></corbaloc>                             | = "corbaloc::" <obj_addr_list>["/"<key_string>]</key_string></obj_addr_list>  |
|   | [, <corbaloc> <corbalocs>]</corbalocs></corbaloc>                             |
| <corbalocs></corbalocs>                           | = "corbalocs::" <obj_addr_list>["/"<key_string>]</key_string></obj_addr_list> |
|   | [, <corbaloc> <corbalocs>]</corbalocs></corbaloc>                             |
| <pre><obj_addr_list></obj_addr_list></pre>        | = [ <obj_addr> ","]* <obj_addr></obj_addr></obj_addr>                         |
| <obj_addr></obj_addr>                             | = <iiop_prot_addr>   <future_prot_addr></future_prot_addr></iiop_prot_addr>   |
|   | = <iiop_id><iiop_addr></iiop_addr></iiop_id>                                  |
| <iiop_id></iiop_id>                               | = "//"   <iiop_prot_token>":"</iiop_prot_token>                               |
| <pre><iiop_prot_toke n=""></iiop_prot_toke></pre> | = "iiop"  |
| <iiop_addr></iiop_addr>                           | = [ <version> <host> [":" <port>]]</port></host></version>                    |
| <host></host>                                     | = DNS-style Host Name   ip_address  |

Table 4-8 (Cont.) BNF Format for URL Elements

| URL Element                    | BNF Format  |  |  |
|--------------------------------|---|--|--|
| <pre><version></version></pre> |   |  |  |
|                                | <pre>= <major> "." <minor> "@"   empty_string</minor></major></pre> |  |  |
|                                |   |  |  |
| <port></port>                  | = number  |  |  |
| <major></major>                | = number  |  |  |
| <minor></minor>                | = number  |  |  |
| <key_string></key_string>      | = <string>   empty_string</string>                                  |  |  |

The following table describes each URL element.

Table 4-9 Descriptions of URL Elements

| URL Element     | Description   |  |
|-----------------|---|--|
| obj_addr_list   | A comma-separated list of protocol ID, version, and address information. This list is used in an implementation-defined manner to address the object. An object may be contacted by any of the addresses and protocols. If a failure occurs using the element, the next element in the comma-separated list will be used. |  |
| obj_addr        | A protocol identifier, version tag, and a protocol specific address. The right-brace "{", left-brace "}", vertical bar " ", slash "/", and comma "," characters are specifically prohibited in this component of the URL.   |  |
| iiop_prot_addr  | An IIOP protocol identifier, version tag, and address containing a DNS-style host name or IP address.   |  |
| iiop_id         | Tokens recognized to identifier, version tag, and address containing a DNS-style host name on IP address.   |  |
| iiop_prot_token | Tokens recognized to indicate an IIOP protocol corbaloc.  |  |
| iiop_addr       | An IIOP protocol token, "iiop".   |  |
| host            | A single address element.   |  |
| version         | A DNS-style host name or IP address. If not present, the local host is assumed.   |  |
| ip_address      | A major and minor version number, separated by "." and followed by "@". If the version is absent, 1.0 is assumed.   |  |
| port            | A numberic IP address (dotted decimal notation).  |  |
| key_string      | A stringified object key that is not NULL-terminated. The key_string uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:        |  |
|                 | ";"   "/"   ":"   "?"   "@"   "&"   "="   "+"   "\$"  |  |
|                 | ","   "-"   "_"   "."   "!"   "~"   "*"   ""   "("   ")"  |  |
|                 | The key_string corresponds to the octet sequence in the object_key member of a GIOP Request or LocateRequest header as defined in the CORBA specification.  |  |

Table 4-9 (Cont.) Descriptions of URL Elements

| URL Element | Description  |  |  |
|-------------|--|--|--|
| string_name | A stringified name with URL escapes as defined in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following: |  |  |
|             | ";"   "/"   ":"   "?"   "@"   "&"   "="   "+"   "\$"   |  |  |
|             | ","   "-"   "_"   "."   "!"   "~"   "*"   ""   "("   ")"   |  |  |

The following are some examples of using the new URL format:

```
corbaloc::555xyz.com:1024,555backup.com:1022,555last.com:1999
corbalocs::555xyz.com:1024,{555backup.com:1022|555last.com:1999}
corbaloc::1.2@555xyz.com:1111
corbalocs::1.1@24.128.122.32:1011,1.0@24.128.122.34
```

As an enhancement to the URL syntax described in the INS submission, Oracle Tuxedo 8.0 or later has extended the syntax to support a list of multiple URLs, each with a different scheme. The following are some examples of the extension:

```
corbalocs::555xyz.com:1024,corbaloc::1.2@555xyz.com:1111
corbalocs::ctxobj:3434,mthd:3434,corbaloc::force:1111
```

In the above example, if the parser reaches the URL <code>corbaloc::force.com:1111</code>, it will reset its internal state as if it had never attempted secure connections and then begins attempting unprotected connections.

## 4.6.5.3 corbaname URL Scheme

The corbaname URL scheme extends the capabilities of the corbaloc scheme to allow URLs to denote entries in a Naming Service. Resolving corbaname URLs does not require a Naming Service implementation in the ORB core. An example of a corbaname URL is:

```
corbaname:555objs.com#a/string/path/to/obj
```

This URL specifies that at host 555objs.com, an object of type NamingContext (with an object key of *NamingService*) can be found, or alternatively, that an agent running at that location will return a reference to a NamingContext. The stringified name a/string/path/to/obj is then used as the argument to the resolve operation on that NamingContext.

A corbaname URL is similar to a corbaloc URL except that a corbaname URL also contains a stringified name that identifies a binding in a naming context. The # character denotes the start of the stringified name.

The BNF syntax for the URL is listed in the following table.



Table 4-10 BNF Syntax for URL

| URL Element                                 | Format  | Description  |
|---|---|--|
| <corbaname></corbaname>                     | = "corbanam e:" <corba loc_obj="">[</corba>                           | corbaloc_obj is a portion of a corbaname URL that identifies the naming context. The syntax is identical to its use in a corbaloc URL. |
|   | "#" <strin<br>g_name&gt;]</strin<br>                                  |  |
| <pre><corbaloc_o bj=""></corbaloc_o></pre>  | <pre><obj_addr_l ist="">["/"<ke y_string="">]</ke></obj_addr_l></pre> | For a description of obj_addr_list, see Table 4-9.   |
| <pre><obj_addr_l ist=""></obj_addr_l></pre> | As defined in a corbaloc URL  | For a description of obj_addr_list, see Table 4-9.   |
| <pre><key_string></key_string></pre>        | As defined in a corbaloc URL  | For a description of key_string, see Table 4-9.  |
| <string_nam e=""></string_nam>              | Stringified<br>Name  <br>empty<br>string                              | For a description of string_name, see Table 4-9.   |

Resolution of a corbaname URL is implemented as a simple extension to corbaloc URL processing. To illustrate the implementation, we will use the following corbaname URL:

```
corbaname:<corbaloc_obj>["#"<string_name>]
```

The resolution process is as follows:

- Construct a corbaloc URL of the form corbaloc::<corbaloc\_obj> from the corbaname URL.
- Convert the corbaloc URL to a naming context object reference by calling CORBA::ORB::string\_to\_object to obtain a CosNaming::NamingContext object.
- 3. Convert < string name > to a CosNaming::Name.
- 4. Invoke the resolve operation on the CosNaming::NamingContext, passing the CosNaming::Name constructed.
- 5. The object reference returned from CosNaming::NamingContext::resolve should be returned to the caller.

By following this resolution process, you eliminate the possibility of returning an object reference for a naming context that does not exist in the Naming Service. One side effect of this approach is that it requires that stubs for the Naming Service be part of the ORB core or that there be an internal mechanism for sending the request for the resolve operation. Because of the complexity, it is recommended that stubs for the Naming Service be embedded within the ORB core.



# 4.6.6 Getting a FactoryFinder Object Reference Using INS

The following code snippet shows an example of how a client application, using INS, gets an object reference to the FactoryFinder object. For a complete code example, see the client application in the University Sample.

```
// utility to get the registrar
static UniversityW::Registrar ptr get registrar(
    CORBA::ORB ptr orb
) {
    // Get the factory finder from the ORB:
    CORBA::Object var v fact finder oref =
        orb -> resolve initial references("FactoryFinder");
    // Narrow the factory finder :
    Tobj::FactoryFinder var v fact finder ref =
        Tobj::FactoryFinder:: narrow(v fact finder oref.in());
    // Use the factory finder to find the
    // university's registrar factory :
    CORBA::Object var v reg fact oref =
        v fact finder ref -> find one factory by id(
            UniversityW:: tc RegistrarFactory -> id()
    // Narrow the registrar factory :
    UniversityW::RegistrarFactory var v reg fact ref =
        UniversityW::RegistrarFactory:: narrow(
            v reg fact oref.in()
    // Return the university's registrar :
    return v reg fact ref -> find registrar();
```

# 4.6.7 Getting a PrincipalAuthenticator Object Reference Using INS

The following code snippet shows an example of how a client application, using INS, gets an object reference to the PrincipalAuthenticator object. For a complete code example, see the client application in the University Sample.

```
// utility to log on to the security system
static SecurityLevel2::PrincipalAuthenticator_ptr logon(
CORBA::ORB_ptr orb,
const char* program_name,
UniversityW::StudentId stu_id
)
{
// Get a Principal Authenticator directly from the ORB:
CORBA::Object_var v_pa_obj =
orb->resolve_initial_references("PrincipalAuthenticator");
// Narrow the Principal Authenticator :
SecurityLevel2::PrincipalAuthenticator_var v_pa =
SecurityLevel2::PrincipalAuthenticator::_narrow(
v pa obj.in());
```

# 4.6.8 Getting a TransactionFactory Object Reference Using INS

As of release 8.0, Oracle Tuxedo CORBA supports the use of the CORBA Transaction Service Interface for beginning transactions. Using the

ORB::resolve\_initial\_references ("FactoryFinder") function, a client gets an object reference to a FactoryFinder. The client then uses the FactoryFinder to get a reference to a TransactionFactory, that it in turn uses to create (begin) a transaction.

The following code snippet shows an example of how a client application, using INS, gets an object reference to the TransactionFactory object. For a complete code example, see the client application in the University Sample.

```
// Get the factory finder from the ORB:
CORBA::Object_var v_fact_finder_oref =
orb->resolve_initial_references("FactoryFinder");
// Narrow the factory finder :
Tobj::FactoryFinder_var v_fact_finder_ref =
Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());
// Get the TransactionFactory from the FactoryFinder
CORBA::Object_var v_txn_fac_oref =
v_fact_finder_ref->find_one_factory_by_id(
"IDL:omg.org/CosTransactions/TransactionFactory:1.0");
// Narrow the TransactionFactory object reference
CosTransactions::TransactionFactory_var v_txn_fac_ref =
CosTransactions::TransactionFactory::_narrow(
v txn fac oref.in());
```

The sequence of events using the INS bootstrapping mechanism is as follows:

- 1. Use ORB::resolve initial references to get a FactoryFinder.
- 2. Use the FactoryFinder to get a TransactionFactory.
- 3. Use the create operation on TransactionFactory to begin a transaction.
- 4. From the Control object returned from the create operation, use the get\_terminator method to get the transaction terminator interface.
- 5. Use the commit or rollback operation on the terminator to end or abort the transaction.

The TransactionFactory returns objects that adhere to the standard CORBA Transaction Service interfaces instead of the Oracle delegated interfaces. This means that a third party ORB can use their ORB's <code>resolve\_initial\_references</code> function to get a reference to a TransactionFactory from an Oracle Tuxedo CORBA server and use stubs generated from standard OMG IDL to act on the instances returned.

Restrictions

### 4.6.8.1 Restrictions

For the Oracle Tuxedo 8.0 release or later, the actions of the TransactionFactory and the client's Current are not coordinated. This means that clients should use one mechanism or the other to control and get status about transactions, not both. Also, only the interfaces and operations listed in The following table are supported. The other operations, as described in the OMG IDL, return the CORBA::NO IMPLEMENT exception.



Table 4-11 Supported INS Interfaces and Operations

| Interface          | Supported Operations           |
|--------------------|--------------------------------|
| TransactionFactory | create                         |
| Control            | get_terminator get_coordinator |
| Terminator         | commit rollback                |
| Coordinator        | get_status<br>rollback_only    |
|                    | get_transaction_name           |



# FactoryFinder Interface

The FactoryFinder interface provides clients with one object reference that serves as the single point of entry into the Oracle Tuxedo domain. The Oracle Tuxedo NameManager provides the mapping of factory names to object references for the FactoryFinder. Multiple FactoryFinders and NameManagers together provide increased availability and reliability. In this release the level of functionality has been extended to support multiple domains.

#### Note:

The NameManager is not a naming service, such as CORBAservices Naming Service, but is merely a vehicle for storing registered factories.

In the Oracle Tuxedo environment, application factory objects are used to create objects that clients interact with to perform their business operations (for example, TellerFactory and Teller). Application factories are generally created during server initialization and are accessed by both remote clients and clients located within the server application.

The FactoryFinder interface and the NameManager services are contained in separate (nonapplication) servers. A set of application programming interfaces (APIs) is provided so that both client and server applications can access and update the factory information.

The support for multiple domains in this release benefits customers that need to scale to a large number of machines or who want to partition their application environment. To support multiple domains, the mechanism used to find factories in an Oracle Tuxedo environment has been enhanced to allow factories in one domain to be visible in another. The visibility of factories in other domains is under the control of the system administrator.

- Capabilities, Limitations, and Requirements
- Functional Description
- C++ Member Functions and Java Methods
- Automation Methods
- Programming Examples

# 5.1 Capabilities, Limitations, and Requirements

During server application initialization, application factories need to be registered with the NameManager. Clients can then be provided with the object reference of a FactoryFinder to allow them to retrieve a factory object reference based on associated names that were created when the factory was registered.

The following functional capabilities, limitations, and requirements apply to this release:

- The FactoryFinder interface is in compliance with the CosLifeCycle::FactoryFinder interface.
- Server applications can register and unregister application factories with the CORBAservices Naming Service.

- Clients can access objects using a single point of entry—the FactoryFinder.
- Clients can construct names for objects using a simplified Oracle scheme made possible by Oracle Tuxedo extensions to the CORBAservices interface or the more general CORBA scheme.
- Multiple FactoryFinders and NameManagers can be used to increase availability and reliability in the event that one FactoryFinder or NameManager should fail.
- Support for multiple domains. Factories in one domain can be configured to be visible in another domain under administrative control.
- Two NameManager services, at a minimum, must be configured, preferably on different
  machines, to maintain the factory-to-object reference mapping across process failures. If
  both NameManagers fail, the master NameManager, which has been keeping a persistent
  journal of the registered factories, recovers the previous state by processing the journal so
  as to re-establish its internal state.
- One NameManager must be designated as the Master and the Master NameManager must be started before the Slave. If the master NameManager is started after one or more Slaves, the Master assumes that it is in recovery mode instead of in initializing mode.

# 5.2 Functional Description

The Oracle Tuxedo CORBA environment promotes the use of the factory design pattern as the primary means for a client to obtain a reference to an object. Through the use of this design pattern, client applications require a mechanism to obtain a reference to an object that acts as a factory for another object. Because the Oracle Tuxedo environment has chosen CORBA as its visible programming model, the mechanism used to locate factories is modeled after the FactoryFinder as described in the CORBAservices Specification, Chapter 6 "Life Cycle Service," December 1997, published by the Object Management Group.

In the CORBA FactoryFinder model, application servers register active factories with a FactoryFinder. When an application server's factory becomes inactive, the application server removes the corresponding registration from the FactoryFinder. Client applications locate factories by querying a FactoryFinder. The client application can control the references to the factory object returned by specifying criteria that is used to select one or more references.

- Locating a FactoryFinder
- Registering a Factory
- Locating a Factory
- Creating Application Factory Keys

# 5.2.1 Locating a FactoryFinder

A client application must obtain a reference to a FactoryFinder before it can begin locating an appropriate factory. To obtain a reference to a FactoryFinder in the domain to which a client application is associated, the client application can use either of two bootstrapping mechanisms:

• Invoke the Tobj\_Bootstrap::resolve\_initial\_references operation with a value of "FactoryFinder". This operation returns a reference to a FactoryFinder that is in the domain to which the client application is currently attached. You should use this mechanism if you are using the Oracle Tuxedo client software. For more information, see the section Tobj\_Bootstrap::resolve\_initial\_references.



• Invoke the CORBA::ORB::resolve\_initial\_references operation with a value of "FactoryFinder". This operation returns a reference to a FactoryFinder that is in the domain to which the client application is currently attached. You should use this mechanism if you are using a third-party client ORB. For more information, see the section CORBA::ORB::resolve\_initial\_references.



The references to the FactoryFinder that are returned to the client application can be references to factory objects that are registered on the same machine as the FactoryFinder, on a different machine than the FactoryFinder, or possibly in a different domain than the FactoryFinder.

# 5.2.2 Registering a Factory

For a client application to be able to obtain a reference to a factory, an application server must register a reference to any factory object for which it provides an implementation with the FactoryFinder. Using the Oracle Tuxedo CORBA TP Framework, the registration of the reference for the factory object can be accomplished using the <code>TP::register\_factory</code> operation, once a reference to a factory object has been created. The reference to the factory object, along with a value that identifies the factory, is passed to this operation. The registration of references to factory objects is typically done as part of initialization of the application (normally as part of the implementation of the operation <code>Server::initialize</code>).

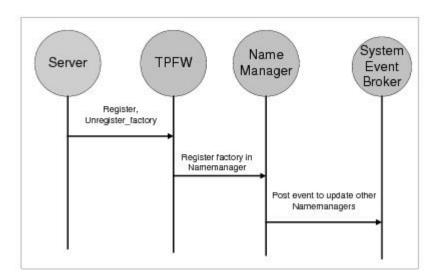


Figure 5-1 Registering a Factory Object

When the server application is shutting down, it must unregister any references to factory objects that it has previously registered in the application server. This is done by passing the same reference to the factory object, along with the corresponding value used to identify the factory, to the TP::unregister\_factory operation. Once unregistered, the reference to the factory object can then be destroyed. The process of unregistering a factory with the FactoryFinder is typically done as part of the implementation of the Server::release operation. For more information about these operations, see the section Server Interface.



#### C++ Mapping

## 5.2.2.1 C++ Mapping

The following code snippet shows the C++ class (static) methods. For more information about these methods, see the sections TP::register\_factory() and TP::unregister\_factory().

```
#include <TP.h>
static void TP::register_factory(
CORBA::Object_ptr factory_or, const char* factory_id);
static void TP::unregister_factory (
CORBA::Object ptr factory or, const char* factory id);
```

The TP.h header file contains the two method declarations. This file must to be included in any server application that wants to use these methods.

A server application generally includes this header file within the application file that contains the methods for application server initialization and release.

# 5.2.3 Locating a Factory

For a client application to request a factory to create a reference to an object, it must first obtain a reference to the factory object. The reference to the factory object is obtained by querying a FactoryFinder with specific selection criteria (see the following figure). The criteria are determined by the format of the particular FactoryFinder interface and method used.

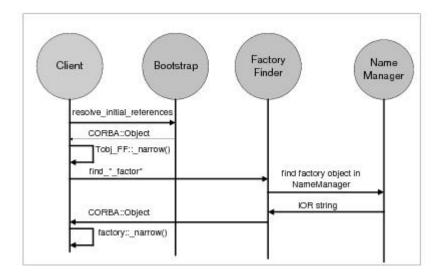


Figure 5-2 Locating a Factory Object

Oracle Tuxedo CORBA extends the <code>CosLifeCycle::FactoryFinder</code> interface by introducing four methods in addition to the <code>find\_factories()</code> method declared for the FactoryFinder. Therefore, using the Tobj extensions, a client can use either the <code>find\_factories()</code> or <code>find\_factories\_by\_id()</code> methods to obtain a list of application factories. A client can also use the <code>find\_one\_factory()</code> or <code>find\_one\_factory\_by\_id()</code> method to obtain a single application factory, and <code>list\_factories()</code> to obtain a list of all registered factories.





You can used the Oracle Tuxedo CORBA extensions to the <code>CosLifeCycle::FactoryFinder</code> interface if you use the Tobj\_Bootstrap object, however, use of the Tobj\_Bootstrap object is not required to locate a factory. If you use CORBA INS, you can use the <code>find\_factories()</code> method provided by the <code>CosLifeCycle::FactoryFinder</code> interface.

The <code>CosLifeCycle::FactoryFinder</code> interface defines a <code>factory\_key</code>, which is a sequence of <code>id</code> and <code>kind</code> strings conforming to the CosNaming Name shown below. The <code>kind</code> field of the NameComponent for all application factories is set to the string <code>FactoryInterface</code> by the TP Framework when an application factory is registered. Applications supply their own value for the <code>id</code> field.

Assuming that the CORBAservices Life Cycle Service modules are contained in their own file (ns.idl and lcs.idl, respectively), only the OMG IDL code for that subset of both files that is relevant for using the Oracle Tuxedo FactoryFinder is shown in the following listings.

- CORBAservices Naming Service Module OMG IDL
- CORBAservices Life Cycle Service Module OMG IDL
- Tobj Module OMG IDL
- Locating Factories in Another Domain
- Why Use Oracle Tuxedo CORBA Extensions?

# 5.2.3.1 CORBAservices Naming Service Module OMG IDL

The following code snippet shows the portions of the ns.idl file that are relevant to the FactoryFinder.

# 5.2.3.2 CORBAservices Life Cycle Service Module OMG IDL

The following code snippet shows the portions of the lcs.idl file that are relevant to the FactoryFinder.

```
// ---- lcs.idl -----
#include "ns.idl"

module CosLifeCycle {
```

# 5.2.3.3 Tobj Module OMG IDL

The following code snippet shows the Tobj Module OMG IDL.

```
// ---- Tobj.idl -----
module Tobj {
        // Constants
        const string FACTORY KIND = "FactoryInterface";
        // Exceptions
        exception CannotProceed {};
        exception InvalidDomain {};
        exception InvalidName {};
        exception RegistrarNotAvailable {};
        // Extension to LifeCycle Service
        struct FactoryComponent {
                CosLifeCycle::Key factory key;
                CosLifeCycle::Factory factory ior;
        typedef sequence < FactoryComponent > FactoryListing;
        interface FactoryFinder: CosLifeCycle::FactoryFinder {
                CosLifeCycle::Factory find one factory(in CosLifeCycle::Key
factory_key)
                raises (CosLifeCycle::NoFactory,
                        Cannot Proceed,
                        RegistrarNotAvailable);
                CosLifeCycle::Factory find one factory by id(in string
factory_id)
                raises (CosLifeCycle::NoFactory,
                        Cannot Proceed,
                        RegistrarNotAvailable);
                CosLifeCycle::Factories find factories by id(in string
factory id)
                raises (CosLifeCycle::NoFactory,
                        Cannot Proceed,
                        RegistrarNotAvailable);
                FactoryListing list factories()
                raises (Cannot Proceed,
                        RegistrarNotAvailable);
        };
};
```

## 5.2.3.4 Locating Factories in Another Domain

Typically, a FactoryFinder returns references to factory objects that are in the same domain as the FactoryFinder itself. However, it is possible to return references to factory objects in domains other than the domain in which a FactoryFinder exists. This can occur if a FactoryFinder contains information about factories that are resident in another domain (see the following figure). A FactoryFinder finds out about these interdomain factory objects through configuration information that describes the location of these other factory objects.

When a FactoryFinder receives a request to locate a factory object, it must first determine if a reference to a factory object that meets the specified criteria exists. If there is registration information for a factory object that matches the criteria, the FactoryFinder must then determine if the factory object is local to the current domain or needs to be imported from another domain. If the factory object is from the local domain, the FactoryFinder returns the reference to the factory object to the client.

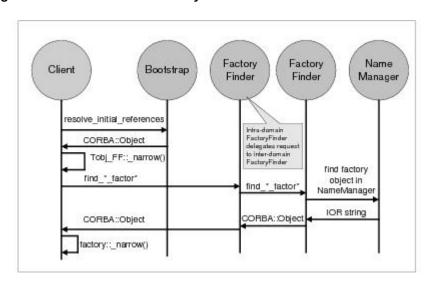


Figure 5-3 Inter-Domain FactoryFinder Interaction

If, on the other hand, the information indicates that the actual factory object is from another domain, the FactoryFinder delegates the request to an interdomain FactoryFinder in the appropriate domain. As a result, only a FactoryFinder in the same domain as the factory object will contain an actual reference to the factory object. The interdomain FactoryFinder is responsible for returning the reference of the factory object to the local FactoryFinder, which subsequently returns it to the client.

# 5.2.3.5 Why Use Oracle Tuxedo CORBA Extensions?

The Oracle Tuxedo software extends the interfaces defined in the CORBAservices specification, Chapter 6 "Life Cycle Service," December 1997, published by the Object Management Group, for the following reasons:

- Although the CORBA-defined approach is powerful and allows various selection criteria, the interface used to query a FactoryFinder can be complicated to use.
- Additionally, if the selection criterion specified by the client application is not specific
  enough, it is possible that more than one reference to a factory object may be returned. If
  this occurs, it is not immediately obvious what a client application should do next.



 Finally, the CORBAservices specification did not specify a standardized mechanism through which an application server is to register a factory object.

Therefore, Oracle Tuxedo extends the interfaces defined in the CORBAservices specification to make using a FactoryFinder easier. The extensions are manifested as refined interfaces to the FactoryFinder that are derived from the interfaces specified in the CORBAservices specification.

# 5.2.4 Creating Application Factory Keys

Two of the five methods provided by the FactoryFinder interface accept CosLifeCycle::Keys, which corresponds to CosNaming::Name. A client must be able to construct these keys.

The CosNaming Specification describes two interfaces that constitute a Names Library interface that can be used to create and manipulate <code>CosLifeCycle::Keys</code>. The pseudo OMG IDL statements for these interfaces is described in the following section.

- Names Library Interface Pseudo OMG IDL
- C++ Mapping
- Java Mapping

# 5.2.4.1 Names Library Interface Pseudo OMG IDL



This information is taken from the *CORBAservices: Common Object Services Specification*, pp. 3-14 to18. Revised Edition: March 31, 1995. Updated: November 1997. Used with permission by OMG.

To allow the representation of names to evolve without affecting existing client applications, it is desirable to hide the representation of names from the client application. Ideally, names themselves would be objects; however, names must be lightweight entities that are efficient to create, manipulate, and transmit. As such, names are presented to programs through the names library.

The names library implements names as pseudo-objects. A client application makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL (to suggest the appropriate language binding). C++ client applications use the same client language bindings for pseudo-IDL (PIDL) as they use for IDL.

Pseudo-object references cannot be passed across OMG IDL interfaces. As described in Chapter 3 of the *CORBAservices: Common Object Services Specification*, in the section "The CosNaming Module," the CORBAservices Naming Service supports the NamingContext OMG IDL interface. The names library supports an operation to convert a library name into a value that can be passed to the name service through the NamingContext interface.

#### Note:

It is not a requirement to use the names library in order to use the CORBAservices Naming Service.

The names library consists of two pseudo-IDL interfaces, the LNameComponent interface and the LName interface, as shown in the following code snippet.

```
interface LNameComponent { // PIDL
        const short MAX LNAME STRLEN = 128;
        exception NotSet {};
        exception OverFlow {};
        string get id
        raises(NotSet);
        void set id(in string i)
        raises (OverFlow);
        string get kind()
        raises (NotSet);
        void set kind(in string k)
        raises (OverFlow);
        void destroy();
};
interface LName { // PIDL
        exception NoComponent {};
        exception OverFlow {};
        exception InvalidName {};
        LName insert component (in unsigned long i,
                in LNameComponent n)
        raises(NoComponent, OverFlow);
        LNameComponent get component (in unsigned long i)
        raises(NoComponent);
        LNameComponent delete component (in unsigned long i)
        raises (NoComponent);
        unsigned long num components();
        boolean equal(in LName ln);
        boolean less than(in LName ln);
        Name to idl form()
        raises(InvalidName);
        void from idl form(in Name n);
        void destroy();
};
LName create lname(); // C/C++
LNameComponent create lname component(); // C/C++
```

- Creating a Library Name Component
- Creating a Library Name
- The LNameComponent Interface
- The LName Interface
- Destroying a Library Name Component Pseudo-Object
- Inserting a Name Component
- Getting the ith Name Component
- Deleting a Name Component
- · Number of Name Components
- · Testing for Equality
- Testing for Order



- Producing an OMG IDL Form
- Translating an IDL Form
- Destroying a Library Name Pseudo-Object

#### 5.2.4.1.1 Creating a Library Name Component

To create a library name component pseudo-object, use the following C/C++ function:

```
LNameComponent create lname component(); // C/C++
```

The returned pseudo-object can then be operated on using the operations shown in the following code snippet.

#### 5.2.4.1.2 Creating a Library Name

To create a library name pseudo-object, use the following C/C++ function:

```
LName create_lname(); // C/C++
```

The returned pseudo-object reference can then be operated on using the operations shown in the following code snippet.

#### 5.2.4.1.3 The LNameComponent Interface

A name component consists of two attributes: identifier and kind. The LNameComponent interface defines the operations associated with these attributes, as follows:

```
string get_id()
raises(NotSet);
void set_id(in string k);
string get_kind()
raises(NotSet);
void set_kind(in string k);
```

#### get id

The get\_id operation returns the identifier attribute's value. If the attribute has not been set, the NotSet exception is raised.

#### set id

The set id operation sets the identifier attribute to the string argument.

#### get\_kind

The get\_kind operation returns the kind attribute's value. If the attribute has not been set, the NotSet exception is raised.

#### set kind

The set kind operation sets the kind attribute to the string argument.

#### 5.2.4.1.4 The LName Interface

The following operations are described in this section:

Destroying a library name component pseudo-object

- Inserting a name component
- Getting the ith name component
- Deleting a name component
- Number of name components
- Testing for equality
- Testing for order
- Producing an OMG IDL form
- Translating an OMG IDL form
- Destroying a library name pseudo-object

#### 5.2.4.1.5 Destroying a Library Name Component Pseudo-Object

The destroy operation destroys library name component pseudo-objects.

```
void destroy();
```

#### 5.2.4.1.6 Inserting a Name Component

A name has one or more components. Each component except the last is used to identify names of subcontexts. (The last component denotes the bound object.) The <code>insert\_component</code> operation inserts a component after position i.

```
LName insert_component(in unsigned long i, in LNameComponent lnc)
raises(NoComponent, OverFlow);
```

If component i-1 is undefined and component i is greater than 1 (one), the insert\_component operation raises the NoComponent exception.

If the library cannot allocate resources for the inserted component, the OverFlow exception is raised.

### 5.2.4.1.7 Getting the ith Name Component

The  $get\_component$  operation returns the i <sup>th</sup> component. The first component is numbered 1 (one).

```
LNameComponent get_component(in unsigned long i)
   raises(NoComponent);
```

If the component does not exist, the NoComponent exception is raised.

### 5.2.4.1.8 Deleting a Name Component

The delete\_component operation removes and returns the i <sup>th</sup> component.

```
LNameComponent delete_component(in unsigned long i)
    raises(NoComponent);
```

If the component does not exist, the NoComponent exception is raised.

After a delete\_component operation has been performed, the compound name has one fewer component and components previously identified as i+1...n are now identified as i...n-1.

#### 5.2.4.1.9 Number of Name Components

The num components operation returns the number of components in a library name.

```
unsigned long num components();
```

#### 5.2.4.1.10 Testing for Equality

The equal operation tests for equality with library name ln.

```
boolean equal(in LName ln);
```

#### 5.2.4.1.11 Testing for Order

The less than operation tests for the order of a library name in relation to library name ln.

```
boolean less than (in LName ln);
```

This operation returns  $\mathtt{TRUE}$  if the library name is less than the library name  $\mathtt{ln}$  passed as an argument. The library implementation defines the ordering on names.

#### 5.2.4.1.12 Producing an OMG IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. Several operations in the NamingContext interface have arguments of an OMG IDL-defined structure, Name. The following PIDL operation on library names produces a structure that can be passed across the OMG IDL request.

```
Name to_idl_form()
raises(InvalidName);
```

If the name is of length 0 (zero), the InvalidName exception is returned.

### 5.2.4.1.13 Translating an IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo-object; therefore, it cannot be passed across the OMG IDL interface for the CORBAservices Naming Service. The NamingContext interface defines operations that return an IDL struct of type  $\mathtt{Name}$ . The following PIDL operation on library names sets the components and  $\mathtt{kind}$  attribute for a library name from a returned OMG IDL defined structure,  $\mathtt{Name}$ .

```
void from idl form(in Name n);
```

#### 5.2.4.1.14 Destroying a Library Name Pseudo-Object

The destroy operation destroys library name pseudo-objects.

```
void destroy();
```

### 5.2.4.2 C++ Mapping

The Names Library pseudo OMG IDL interface maps to the C++ classes shown in the following listing, which can be found in the NamesLib.h header file.

Two Oracle Tuxedo extensions to CORBA are included to support scalability. Specifically, the LNameComponent::set\_id() and LNameComponent::set\_kind() methods raise an OverFlow exception if the length of the input string exceeds MAX\_LNAME\_STRLEN. This length coincides with the maximum length of the Oracle Tuxedo object ID (OID) and interface name. For a detailed description of the Library Name class, see the section Names Library Interface Pseudo OMG IDL.

```
const short MAX LNAME STRLEN = 128;
class LNameComponent {
public:
class NotSet{ };
class OverFlow{ };
static LNameComponent* create_lname_component();
void destroy();
const char* get id() const throw (NotSet);
void set id(const char* i) throw (OverFlow);
const char* get kind() const throw (NotSet);
void set_kind(const char* k) throw (OverFlow);
class LName {
public:
class NoComponent{ };
class OverFlow{ };
class InvalidName{ };
static LName* create lname();
void destroy();
LName* insert component (const unsigned long i,
LNameComponent* n)
throw (NoComponent, OverFlow);
const LNameComponent* get component(
const unsigned long i) const
throw (NoComponent);
const LNameComponent* delete component(
const unsigned long i)
throw (NoComponent);
unsigned long num components() const;
CORBA::Boolean equal(const LName* ln) const;
CORBA::Boolean less than (
const LName* ln) const; // not implemented
CosNaming::Name* to_idl_form()
throw (InvalidName);
```



```
void from_idl_form(const CosNaming::Name& n);
};
```

### 5.2.4.3 Java Mapping

The Names Library pseudo OMG IDL interface maps to the Java classes contained in the com.beasys.Tobj package, shown in the following code snippet. All exceptions are contained in the same package.

For a detailed description of the Library Name class, refer to Chapter 3 in the *CORBAservices: Common Object Services Specification*.

```
public class LNameComponent {
    public static LNameComponent create lname component();
    public static final short MAX LNAME STRING = 128;
    public void destroy();
   public String get id() throws NotSet;
    public void set id(String i) throws OverFlow;
   public String get kind() throws NotSet;
   public void set kind(String k) throws OverFlow;
};
   public class LName {
   public static LName create lname();
   public void destroy();
    public LName insert component(long i, LNameComponent n)
       throws NoComponent, OverFlow;
    public LNameComponent get component(long i)
       throws NoComponent;
    public LNameComponent delete component (long i)
       throws NoComponent;
    public long num components();
    public boolean equal(LName ln);
    public boolean less than (LName ln); // not implemented
    public org.omg.CosNaming.NameComponent[] to idl form()
       throws InvalidName;
    public void from idl form(org.omg.CosNaming.NameComponent[] nr);
};
```

### 5.3 C++ Member Functions and Java Methods

This section describes the FactoryFinder C++ member functions and Java methods.



All FactoryFinder member functions, except the <code>less\_than</code> member function in LName, are implemented in both C++ and Java.

The following methods are described in this section:

- CosLifeCycle::FactoryFinder::find\_factories
- Tobj::Factoryfinder::find\_one\_factory

- Tobj::Factoryfinder::find one factory by id
- Tobj::Factoryfinder::find factories by id
- Tobj::Factoryfinder::list factories

#### Note:

The CosLifeCycle::FactoryFinder::find\_factories method is the standard CORBA CosLifeCycle method. The four Tobj methods are extensions to the CosLifeCycle interface and, therefore, inherit the attributes of the CosLifeCycle interface.

- CosLifeCycle::FactoryFinder::find\_factories
- Tobj::FactoryFinder::find\_one\_factory
- Tobj::FactoryFinder::find\_one\_factory\_by\_id
- Tobj::FactoryFinder::find\_factories\_by\_id
- Tobj::Factoryfinder::list\_factories

### 5.3.1 CosLifeCycle::FactoryFinder::find\_factories

- Synopsis
- C++ Mapping
- Java Mapping
- Parameter
- Exception
- Description
- Return Values

### 5.3.1.1 Synopsis

Obtains a sequence of factory object references.

### 5.3.1.2 C++ Mapping

```
CosLifeCycle::Factories *
CORBA::Object_ptr CosLifeCycle::FactoryFinder::find_factories(
const CosNaming::Name& factory_key)
throw (CosLifeCycle::NoFactory);
```

### 5.3.1.3 Java Mapping

```
import org.omg.CosLifeCycle.*;
public org.omg.CORBA.Object[] find_factories(
org.omg.CosNaming.NameComponent[] factory_key)
throws org.omg.CosLifeCycle.NoFactory;
```

#### 5.3.1.4 Parameter

#### factory key

This parameter is an unbounded sequence of NameComponents (tuple of <id, kind> pairs) that uniquely identifies a factory object reference.

A NameComponent is defined as a having two members: an id and a kind, both of type string. The id field is used to represent the identity of factory object. The kind field is used to indicate how the value of the id field should be interpreted.

References to factory object registered using the operation TP::register\_factory will have a kind value of "FactoryInterface".

### 5.3.1.5 Exception

#### CORBA::BAD PARAM

Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory key is specified.

#### CosLifeCycle::NoFactory

Indicates that there are no factories registered that match the information in the factory\_key parameter.

### 5.3.1.6 Description

The find\_factories method is called by an application to obtain a sequence of factory object references. The operation is passed a key used to identify the desired factory. The key is a name, as defined by the CORBAservices Naming service. More than one factory may match the key, and, if that is the case, the FactoryFinder returns a sequence of factories.

The scope of the key is the FactoryFinder. The FactoryFinder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factories or objects they create.

Key values are considered equal if they are of equal length (same number of elements in the sequence), and if every NameComponent value in the key matches the corresponding NameComponent value at the exact same location in the key that was specified when the reference to the factory object was registered.

#### 5.3.1.7 Return Values

An unbounded sequence of references to factory objects that match the information specified as the value of the <code>factory\_key</code> parameter. In C++, the method returns a sequence of object references of type <code>CosLifeCycle::Factory</code>. In Java, the method returns an unbounded array of object references of type <code>org.omg.CORBA.Object</code>.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

### 5.3.2 Tobj::FactoryFinder::find\_one\_factory

Synopsis



- C++ Mapping
- Java Mapping
- Parameter
- Exceptions
- Description
- Return Values

### **5.3.2.1** Synopsis

Obtains a reference to a single factory object.

### 5.3.2.2 C++ Mapping

```
virtual CosLifeCycle::Factory_ptr
find one factory( const CosNaming::Name& factory key) = 0;
```

### 5.3.2.3 Java Mapping

```
public org.omg.CORBA.Object
find_one_factory( org.omg.CosNaming.NameComponent[] factory_key)
throws
org.omg.CosLifeCycle.NoFactory,
com.beasys.Tobj.CannotProceed,
com.beasys.Tobj.RegistrarNotAvailable;
```

#### 5.3.2.4 Parameter

#### factory key

This parameter is an unbounded sequence of NameComponents (tuple of <id, kind> pairs) that uniquely identifies a factory object reference.

A NameComponent is defined as a having two members: an id and a kind, both of type string. The id field is used to represent the identity of factory object. The kind field is used to indicate how the value of the id field should be interpreted.

References to factory object registered using the operation TP::register\_factory will have a kind value of "FactoryInterface".

### 5.3.2.5 Exceptions

#### CORBA::BAD PARAM

Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory key is specified.

#### CosLifeCycle::NoFactory

Indicates that there are no factories registered that match the information in the factory\_key parameter.



#### Tobj::CannotProceed

Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object.

Error information is written to the user log.

#### Tobj::RegistrarNotAvailable

Indicates that the FactoryFinder could not communicate with the NameManager.

Error information is written to the user log.

#### 5.3.2.6 Description

The find\_one\_factory method is called by an application to obtain a reference to a single factory object whose key matches the value of the key specified as input to the method. If more than one factory object is registered with the specified key, the FactoryFinder selects one factory object based on the FactoryFinder's load balancing scheme. As a result, invoking the find\_one\_factory method multiple times using the same key may return different object references.

The scope of the key is the FactoryFinder. The FactoryFinder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factory or objects they create.

Key values are considered equal if they are of equal length (same number of elements in the sequence), and if every NameComponent value in the key matches the corresponding NameComponent value at the exact same location in the key that was specified when the reference to the factory object was registered.

#### 5.3.2.7 Return Values

An object reference for a factory object. In C++, the method returns an object reference of type CosLifeCycle::Factory. In Java, the method returns an object reference of type org.omg.CORBA.Object.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

### 5.3.3 Tobj::FactoryFinder::find\_one\_factory\_by\_id

- Synopsis
- C++ Mapping
- Java Mapping
- Parameter
- Exceptions
- Description
- Return Values

### 5.3.3.1 Synopsis

Obtains a reference to a single factory object.

### 5.3.3.2 C++ Mapping

```
virtual CosLifeCycle::Factory_ptr
find_one_factory_by_id( const char * factory_id) = 0;
```

### 5.3.3.3 Java Mapping

```
public org.omg.CORBA.Object
find_one_factory_by_id( java.lang.String factory_id)
throws
org.omg.CosLifeCycle.NoFactory,
com.beasys.Tobj.CannotProceed,
com.beasys.Tobj.RegistrarNotAvailable;
```

#### 5.3.3.4 Parameter

#### factory id

A NULL-terminated string that contains a value that is used to identify the registered factory object to be found.

The value of the factory\_id parameter is used as the value of the id field of a NameComponent that has a kind field with the value "FactoryInterface" when comparing against registered references for factory objects.

### 5.3.3.5 Exceptions

#### CORBA::BAD PARAM

Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory key is specified.

#### CosLifeCycle::NoFactory

Indicates that there are no factories registered that match the information in the factory\_key parameter.

#### Tobj::CannotProceed

Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object.

Error information is written to the user log.

#### Tobj::RegistrarNotAvailable

Indicates that the FactoryFinder could not communicate with the NameManager.

Error information is written to the user log.

### 5.3.3.6 Description

The <code>find\_one\_factory\_by\_id</code> method is called by an application to obtain a reference to a single factory object whose registration ID matches the value of the ID specified as input to the method. If more than one factory object is registered with the specified ID, the FactoryFinder selects one factory object based on the FactoryFinder's load balancing scheme. As a result,

invoking the find\_one\_factory\_by\_id operation multiple times using the same ID may return different object references.

The <code>find\_one\_factory\_by\_id</code> method behaves the same as the <code>find\_one\_factory</code> operation that was passed a key that contains a single NameComponent with an <code>id</code> field that contains the same value as the <code>factory\_id</code> parameter and a <code>kind</code> field that contains the value <code>"FactoryInterface"</code>.

The registered identifier for a factory is considered equal to the value of the <code>factory\_id</code> parameter if the result of constructing a <code>CosLifeCycle::Key</code> structure containing a single <code>NameComponent</code> that has the <code>factory\_id</code> parameter as the value of the <code>id</code> field and the value <code>"FactoryInterface"</code> as the value of the <code>kind</code> field. The values must match exactly in all respects (case, location, etc.).

#### 5.3.3.7 Return Values

An object reference for a factory object. In C++, the method returns an object reference of type CosLifeCycle::Factory. In Java, the method returns an object reference of type org.omg.CORBA.Object.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

### 5.3.4 Tobj::FactoryFinder::find\_factories\_by\_id

- Synopsis
- C++ Mapping
- Java Mapping
- Parameter
- Exceptions
- Description
- Return Values

### 5.3.4.1 Synopsis

Obtains a sequence of one or more factory object references.

### 5.3.4.2 C++ Mapping

```
virtual CosLifeCycle::Factories *
find factories by id( const char * factory id) = 0;
```

### 5.3.4.3 Java Mapping

```
public org.omg.CORBA.Object[]
find_factories_by_id( java.lang.String factory_id)
throws
org.omg.CosLifeCycle.NoFactory,
com.beasys.Tobj.CannotProceed,
com.beasys.Tobj.RegistrarNotAvailable;
```

#### 5.3.4.4 Parameter

#### factory id

A NULL-terminated string that contains a value that is used to identify the registered factory object to be found.

The value of the factory\_id parameter is used as the value of the id field of a NameComponent that has a kind field with the value "FactoryInterface" when comparing against registered references for factory objects.

### 5.3.4.5 Exceptions

#### CORBA::BAD PARAM

Indicates that the value of an input parameter has an inappropriate value or is invalid. Of particular importance, the exception is raised if no value or a NULL value for the parameter factory key is specified.

#### CosLifeCycle::NoFactory

Indicates that there are no factories registered that match the information in the factory\_key parameter.

#### Tobj::CannotProceed

Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object.

Error information is written to the user log.

#### Tobj::RegistrarNotAvailable

Indicates that the FactoryFinder could not communicate with the NameManager.

Error information is written to the user log.

### 5.3.4.6 Description

The <code>find\_factories\_by\_id</code> method is called by an application to obtain a sequence of one or more factory object references. The method is passed a NULL-terminated string that contains the identifier of the factory to be located. If more than one factory object is registered with the specified ID, the FactoryFinder will return a list of object references for the matching registered factory objects.

The  $find_factories_by_id$  method behaves the same as the  $find_factory$  operation that was passed a key that contains a single NameComponent with an id field that contains the same value as the  $factory_id$  parameter and a kind field that contains the value "FactoryInterface".

The registered identifier for a factory is considered equal to the value of the <code>factory\_id</code> parameter if the result of constructing a <code>CosLifeCycle::Key</code> structure containing a single NameComponent that has the <code>factory\_id</code> parameter as the value of the <code>id</code> field and the value <code>"FactoryInterface"</code> as the value of the <code>kind</code> field. The values must match exactly in all respects (case, location, etc.).



#### 5.3.4.7 Return Values

An unbounded sequence of references to factory objects that match the information specified as the value of the <code>factory\_key</code> parameter. In C++, the method returns a sequence of object references of type <code>CosLifeCycle::Factory</code>. In Java, the method returns an unbounded array of object references of type <code>org.omg.CORBA.Object</code>.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

### 5.3.5 Tobj::Factoryfinder::list factories

- Synopsis
- C++ Mapping
- Java Mapping
- Exception
- Description
- Return Values

### 5.3.5.1 Synopsis

Obtains a list of factory objects currently registered with the FactoryFinder.

#### 5.3.5.2 C++ Mapping

```
virtual FactoryListing * list factories() = 0;
```

### 5.3.5.3 Java Mapping

```
public com.beasys.Tobj.FactoryComponent[] list_factories()
throws
com.beasys.Tobj.CannotProceed,
com.beasys.Tobj.RegistrarNotAvailable;
```

### 5.3.5.4 Exception

#### Tobj::CannotProceed

Indicates that the FactoryFinder or NameManager encountered an internal error while attempting to locate a reference for a factory object.

Error information is written to the user log.

#### Tobj::RegistrarNotAvailable

Indicates that the FactoryFinder could not communicate with the NameManager.

Error information is written to the user log.

### 5.3.5.5 Description

The <code>list\_factories</code> method is called by an application to obtain a list of the factory objects currently registered with the FactoryFinder. The method returns both the key used to register the factory, as well as a reference to the factory object.

The number of factories returned by <code>list\_factories</code> will be one more than the ones registered by the user. For example, if the user registered four factories then the number of factories returned by <code>list\_factories</code> will be five.



This change in behavior is because the OMG Transaction Service specification version 1.1 in section 2.1.2 specifies that the Transaction Factory is located using the FactoryFinder interface of the Life Cycle Service. Hence the Transaction factory is registered internally by the product with the FactoryFinder.

#### 5.3.5.6 Return Values

An unbounded sequence of Tobj::FactoryComponent. Each occurrence of a Tobj::FactoryComponent in the sequence contains a reference to the registered factory object, as well as the CosLifeCycle::Key that was used to register that factory object.

If the operation raises an exception, the return value is invalid and does not need to be released by the caller.

### 5.4 Automation Methods

This section describes the DITobj\_FactoryFinder Automation methods.

- DITobj\_FactoryFinder.find\_one\_factory
- DITobj\_FactoryFinder.find\_one\_factory\_by\_id
- DITobj\_FactoryFinder.find\_factories\_by\_id
- DITobj FactoryFinder.find factories
- DITobj\_FactoryFinder.list\_factories

### 5.4.1 DITobj FactoryFinder.find one factory

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameters
- Exceptions
- Description
- Return Values



### 5.4.1.1 Synopsis

Obtains a single application factory.

### 5.4.1.2 MIDL Mapping

```
HRESULT find_one_factory(
[in] VARIANT factory_key,
[in,out,optional] VARIANT* exceptionInfo,
[out,retval] IDispatch** returnValue);
```

### 5.4.1.3 Automation Mapping

```
Function find one factory(factory key, [exceptionInfo]) As Object
```

#### 5.4.1.4 Parameters

#### factory\_key

This parameter contains a safe array of DICosNaming\_NameComponent (<id, kind> value pairs) that uniquely identifies a factory object reference.

#### exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.

### 5.4.1.5 Exceptions

#### NoFactory

This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory key.

#### CannotProceed

This exception is raised if the FactoryFinder or CORBAservices Naming Service encounter an internal error during the search with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or CORBAservices Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBAservices Naming Service has terminated and there is another CORBAservices Naming Service running, start a new CORBAservices Naming Service. If no naming services servers are running, restart the application.

#### RegistrarNotAvailable

This exception is raised if the FactoryFinder object cannot locate the CORBAservices Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.

### 5.4.1.6 Description

This member function instructs the FactoryFinder to return one application factory object reference whose key matches the input <code>factory\_key</code>. To accomplish this, the member function performs an equality match; that is, every NameComponent <id, kind> pair in the input <code>factory\_key</code> must exactly match each <d, kind> pair in the application factory's key. If multiple

factory keys contain the input factory\_key, the FactoryFinder selects one factory key, based on an internally defined load balancing scheme. Invoking find\_one\_factory multiple times using the same id may return different object references.

#### 5.4.1.7 Return Values

Returns a reference to an interface pointer for the application factory.

# 5.4.2 DITobj\_FactoryFinder.find\_one\_factory\_by\_id

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameters
- Exceptions
- Description
- Return Values

### 5.4.2.1 Synopsis

Obtains a single application factory.

### 5.4.2.2 MIDL Mapping

```
HRESULT find_one_factory_by_id(
[in] BSTR factory_id,
[in,out,optional] VARIANT* exceptionInfo,
[out,retval] IDispatch** returnValue);
```

### 5.4.2.3 Automation Mapping

```
Function find_one_factory_by_id(factory_id As String,
[exceptionInfo]) As Object
```

#### 5.4.2.4 Parameters

#### factory id

This parameter represents a string identifier that is used to identify the kind or type of application factory. For some suggestions as to the composition of this string, see *Creating CORBA Server Applications*.

#### exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.



### 5.4.2.5 Exceptions

#### NoFactory

This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory id.

#### CannotProceed

This exception is raised if the FactoryFinder or CORBAservices Naming Service encounter an internal error during the search, with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the CORBAservices Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBAservices Naming Service has terminated and there is another CORBAservices Naming Service running, start a new CORBAservices Naming Service. If there are no naming services running, restart the application.

#### RegistrarNotAvailable

This exception is raised if the FactoryFinder object cannot locate the CORBAservices Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming service servers are running, restart the application.

### 5.4.2.6 Description

This member function instructs the FactoryFinder to return one application factory object reference whose <code>id</code> in the key matches the method's input <code>factory\_id</code>. To accomplish this, the member function performs an equality match (that is, the input <code>factory\_id</code> must exactly match the <code>id</code> in the <code><id</code>,kind> pair in the application factory's key). If multiple factory keys contain the input <code>factory\_id</code>, the FactoryFinder selects one factory key, based on an internally defined load balancing scheme. Invoking <code>find\_one\_factory\_by\_id</code> multiple times using the same <code>id</code> may return different object references.

#### 5.4.2.7 Return Values

Returns a reference to an interface pointer for the application factory.

### 5.4.3 DITobj FactoryFinder.find factories by id

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameters
- Exceptions
- Description
- Return Values

### 5.4.3.1 Synopsis

Obtains a list of application factories.



### 5.4.3.2 MIDL Mapping

```
HRESULT find_factories_by_id(
[in] BSTR factory_id,
[in,out,optional] VARIANT* exceptionInfo,
[out,retval] VARIANT* returnValue);
```

### 5.4.3.3 Automation Mapping

```
Function find_factories_by_id(factory_id As String,
[exceptionInfo])
```

#### 5.4.3.4 Parameters

#### factory id

This parameter represents a string identifier that will be used to identify the kind or type of application factory. The *Creating CORBA Client Applications* online document provides some suggestions as to the composition of this string.

#### exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.

### 5.4.3.5 Exceptions

#### NoFactory

This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory key or factory id.

#### CannotProceed

This exception is raised if the FactoryFinder or CORBAservices Naming Service encounter an internal error during the search with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or CORBAservices Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBAservices Naming Service has terminated and there is another CORBAservices Naming Service running, start a new CORBAservices Naming Service. If no naming services servers are running, restart the application.

#### RegistrarNotAvailable

This exception is raised if the FactoryFinder object cannot locate the CORBAservices Naming Service object. Notify the operations staff immediately if this exception is raised. If no naming services servers are running, restart the application.

### 5.4.3.6 Description

This member function instructs the FactoryFinder to return a list of application factory object references whose id in the keys match the method's input factory\_id. To accomplish this, the member function performs an equality match (that is, the input factory\_id must exactly match each id in the <id,kind> pair in the application factory's keys).



#### 5.4.3.7 Return Values

Returns a variant containing an array of interface pointers to application factories.

### 5.4.4 DITobj\_FactoryFinder.find\_factories

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameters
- Exception
- Description
- Return Values

### 5.4.4.1 Synopsis

Obtains a list of application factories.

### 5.4.4.2 MIDL Mapping

```
HRESULT find_factories(
[in] VARIANT factory_key,
[in,out,optional] VARIANT* exceptionInfo,
[out,retval] VARIANT* returnValue);
```

### 5.4.4.3 Automation Mapping

Function find\_factories(factory\_key, [exceptionInfo])

#### 5.4.4.4 Parameters

#### factory key

This parameter contains a safe array of DICosNaming\_NameComponents (<d, kind> value pairs) that uniquely identifies a factory object reference.

#### exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.

### 5.4.4.5 Exception

#### NoFactory

This exception is raised if the FactoryFinder cannot find an application factory object reference that corresponds to the input factory key.

### 5.4.4.6 Description

The find\_factories method instructs the FactoryFinder to return a list of server application factory object references whose keys match the method's input key. The Oracle Tuxedo

system assumes that an equality match is to be performed. This means that for the two sequences of <d,kind> pairs (those corresponding to the input key and those in the application factory's keys), each are of equal length; for every pair in one sequence, there is an identical pair in the other.

### 5.4.4.7 Return Values

Returns a variant containing an array of interface pointers to application factories.

### 5.4.5 DITobj\_FactoryFinder.list\_factories

- Synopsis
- MIDL Mapping
- Automation Mapping
- Parameter
- Exception
- Description
- Return Values

### 5.4.5.1 Synopsis

Lists all of the application factory names and object references.

### 5.4.5.2 MIDL Mapping

```
HRESULT list_factories(
[in,out,optional] VARIANT* exceptionInfo,
[out,retval] VARIANT* returnValue);
```

### 5.4.5.3 Automation Mapping

```
Function list factories([exceptionInfo])
```

#### 5.4.5.4 Parameter

#### exceptionInfo

An optional input argument that enables the application to get additional exception data if an error occurred.

### 5.4.5.5 Exception

#### CannotProceed

This exception is raised if the FactoryFinder or the CORBAservices Naming Service encounter an internal error during the search with the error being written to the user log (ULOG). Notify the operations staff immediately if this exception is raised. Depending on the severity of the internal error, the server running the FactoryFinder or the CORBAservices Naming Service may have terminated. If a FactoryFinder service has terminated, start a new FactoryFinder service. If a CORBAservices Naming Service has terminated and there is another



CORBAservices Naming Service running, start a new CORBAservices Naming Service. If there are no naming service servers running, restart the application.

#### RegistrarNotAvailable

This exception is raised if the FactoryFinder object cannot locate the CORBAservices Naming Service object. Notify the operations staff immediately if this exception is raised. It is possible that no naming service servers are running. Restart the application.

### 5.4.5.6 Description

This method instructs the FactoryFinder to return a list containing all of the factory keys and associated object references for application factories registered with the CORBAservices Naming Service.

#### 5.4.5.7 Return Values

Returns a variant containing an array of DITobj\_FactoryComponent objects. The FactoryComponent object consists of a variant containing an array of DICosNaming NameComponent objects and an interface pointer to the application factory.

# 5.5 Programming Examples

This section describes how to program using the FactoryFinder interface.



Remember to check for exceptions in your code.

- Using the FactoryFinder Object
- Using Extensions to the FactoryFinder Object

## 5.5.1 Using the FactoryFinder Object

A FactoryFinder object is used by programmers to locate a reference to a factory object. The FactoryFinder object provides operations to obtain one or more references to factory objects based on the criteria specified.

There can be more than one FactoryFinder object in a process address space. Multiple references to a FactoryFinder object must be supported. A FactoryFinder object is semi-stateful in that it maintains state about the association between FactoryFinder objects within a domain and a particular IIOP Server Listener/Handler (ISL/ISH) through which to access the domain.

All FactoryFinder objects support the <code>CosLifeCycle::FactoryFinder</code> interface as defined in CORBAservices Specification, Chapter 6 "Life Cycle Service," December 1997, published by the Object Management Group. The interface contains one operation that is used to obtain one or more references to factory objects that meet the criteria specified.

- Registering a Reference to a Factory Object
- Obtaining a Reference to a FactoryFinder Object Using the CosLifeCycle::FactoryFinder Interface
- Obtaining a Reference to a FactoryFinder Object Using the Extensions Bootstrap object



### 5.5.1.1 Registering a Reference to a Factory Object

The following code snippet shows how to use the TP Framework interface to register a reference to a factory object with a FactoryFinder.

```
// Server Application: Registering a factory.
// C++ Example.

TP::register factory( factory obj.in(), "TellerFactory");
```

# 5.5.1.2 Obtaining a Reference to a FactoryFinder Object Using the CosLifeCycle::FactoryFinder Interface

The following code snippet shows how to use of the CORBA-compliant interface to obtain one or more references to factory objects.

```
// Client Application: Obtaining the object reference
// to factory objects.
CosLifeCycle::Key_var factory_key = new CosLifeCycle::Key();
factory_key ->length(1);
factory_key[0].id = string_dupalloc("strlen("TellerFactory") +1);
factory_key[0].kind = string_dupalloc(
strlen(""FactoryInterface") + 1);
strcpy(factory_key[0].id, ""TellerFactory");
strcpy(facory_key[0].kind, "FactoryInterface");
CosLifeCycle::Factories_var * flp = ff_np ->
find_factories(factory_key.in());
```

# 5.5.1.3 Obtaining a Reference to a FactoryFinder Object Using the Extensions Bootstrap object

The following code snippet shows how to use of the Oracle Tuxedo extensions Bootstrap object to obtain a reference to a FactoryFinder object.

```
// Client Application: Finding one factory using the Tobj
// approach.
Tobj_Bootstrap * bsp = new Tobj_Bootstrap(
orb_ptr.in(), host_port);
CORBA::Object_varptr ff_op = bsp ->
resolve_initial_references( "FactoryFinder");
Tobj::FactoryFinder_ptrvar ff_np =
Tobj::FactoryFinder::_narrow( ff_op);
```



You can used the Oracle Tuxedo CORBA extensions to the <code>CosLifeCycle::FactoryFinder</code> interface if you use the Tobj\_Bootstrap object, however, use of the Tobj\_Bootstrap object is not required to locate a factory. If you use CORBA INS, you can use the <code>find\_factories()</code> method provided by the <code>CosLifeCycle::FactoryFinder</code> interface.

### 5.5.2 Using Extensions to the FactoryFinder Object

Oracle Tuxedo extends the FactoryFinder object with functionality to support similar capabilities to those provided by the operations defined by CORBA, but with a much simpler and more restrictive signature. The enhanced functionality is provided by defining the <code>Tobj::FactoryFinder</code> interface. The operations defined for the <code>Tobj::FactoryFinder</code> interface are intended to provide a focused, simplified form of the equivalent capability defined by CORBA. An application developer can choose to use the CORBA-defined or Oracle Tuxedo extensions when developing an application. The interface <code>Tobj::FactoryFinder</code> is derived from the <code>CoslifeCycle::FactoryFinder</code> interface.

Oracle Tuxedo extensions to the FactoryFinder object adhere to all the same rules as the FactoryFinder object defined in the CORBAservices Specification, Chapter 6 "Life Cycle Service," December 1997, published by the Object Management Group.

The implementation of the extended FactoryFinder object requires users to supply either a CosLifeCycle::Key, as in the CORBA-defined CosLifeCycle::FactoryFinder interface, or a NULL-terminated string containing the identifier of a factory object to be located.

- Obtaining One Factory Using Tobj::FactoryFinder
- Obtaining One or More Factories Using Tobj::FactoryFinder

### 5.5.2.1 Obtaining One Factory Using Tobj::FactoryFinder

The following code snippet shows how to use the Oracle Tuxedo extensions interface to obtain one reference to a factory object based on an identifier.

```
CosLifeCycle::Factory_ptrvar fp_obj = ff_np ->
find one factory by id( "TellerFactory" );
```

### 5.5.2.2 Obtaining One or More Factories Using Tobj::FactoryFinder

The following code snippet shows how to use the Oracle Tuxedo extensions to obtain one or more references to factory objects based on an identifier.

```
CosLifeCycle::Factories * _var flp = ff_np ->
find_factories_by_id( "TellerFactory" );
```

# **Security Service**

For a detailed discussion of Security, see *Using Security in CORBA Applications*. This document provides an introduction to crytography and other concepts associated with the Oracle Tuxedo security features, a description of how to secure your Oracle Tuxedo applications using the security features, and a guide to the use of the application programming interfaces (APIs) in the Security Service.

A PDF file of Using Security in CORBA Applications is also provided in the online documentation.



# **Transactions Service**

For a detailed discussion of Transactions, see *Using CORBA Transactions*. This document provides an introduction to transactions, a description of the application programming interfaces (APIs), and a guide to the use of the application programming interfaces (APIs) to develop applications.

A PDF file of Using CORBA Transactions is also provided in the online documentation.



# **Notification Service**

For a detailed discussion of the Notification Service, see *Using the CORBA Notification Service*. This document provides an introduction to the Notification Service, a description of the application programming interfaces (APIs), and a guide to the use of the APIs to develop applications.

A PDF file of Using the CORBA Notification Service is also provided in the online documentation.



# Request-Level Interceptors

For a detailed discussion of request-level interceptors, see *Using CORBA Request-Level Interceptors*. This document provides an introduction to request-level interceptors, a description of the application programming interfaces (APIs), and a guide to the use of the APIs to implement request-level interceptors.

A HTML file of Using CORBA Request-Level Interceptors is also provided in the online documentation.



# **CORBA Interface Repository Interfaces**

The Oracle Tuxedo CORBA Interface Repository contains the interface descriptions of the CORBA objects that are implemented within the Oracle Tuxedo domain.

The Interface Repository is based on the CORBA definition of an Interface Repository. It offers a proper subset of the interfaces defined by CORBA; that is, the APIs that are exposed to programmers are implemented as defined by the *Common Object Request Broker: Architecture and Specification* Revision 2.4. However, not all interfaces are supported. In general, the interfaces required to read from the Interface Repository are supported, but the interfaces required to write to the Interface Repository are not. Additionally, not all TypeCode interfaces are supported.

Administration of the Interface Repository is done using tools specific to the Oracle Tuxedo software. These tools allow the system administrator to create an Interface Repository, populate it with definitions specified in Object Management Group Interface Definition Language (OMG IDL), and then delete interfaces. Additionally, an administrator may need to configure the system to include an Interface Repository server. For a description of the Interface Repository administration commands, see the Oracle Tuxedo Command Reference and Oracle® Tuxedo Application Configuration Guide.

Several abstract interfaces are used as base interfaces for other objects in the Interface Repository. A common set of operations is used to locate objects within the Interface Repository. These operations are defined in the abstract interfaces IRObject, Container, and Contained described in this chapter. All Interface Repository objects inherit from the IRObject interface, which provides an operation for identifying the actual type of the object. Objects that are containers inherit navigation operations from the Container interface. Objects that are contained by other objects inherit navigation operations from the Contained interface. The IDLType interface is inherited by all Interface Repository objects that represent OMG IDL types, including interfaces, typedefs, and anonymous types. The TypedefDef interface is inherited by all named noninterface types.

The IRObject, Contained, Container, IDLType, and TypedefDef interfaces are not instantiable.

All string data in the Interface Repository are encoded as defined by the ISO 8859-1 character set.



The Write interface is not documented in this chapter because the Oracle Tuxedo software supports only read access to the Interface Repository. Any attempt to use the Write interface to the Interface Repository will raise the exception CORBA::NO IMPLEMENT.



#### Note:

Most of the information in this chapter is taken from Chapter 10 of the *Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, February 2001. The OMG information has been modified as required to describe the Oracle Tuxedo CORBA implementation of the Interface Repository interfaces. Used with permission of the OMG. The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- to help implement/run third party Java ORB libraries, and
- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

This chapter describes the Oracle Tuxedo CORBA Interface Repository interfaces.

- Structure and Usage
- Building Client Applications
- Getting Initial References to the InterfaceRepository Object
- Interface Repository Interfaces

# 10.1 Structure and Usage

The Interface Repository consists of two distinct components: the database and the server. The server performs operations on the database.

The Interface Repository database is created and populated using the idl2ir administrative command. For a description of this command, see the Oracle Tuxedo Command Reference and Oracle® Tuxedo Application Configuration Guide. From the programmer's point of view, there is no write access to the Interface Repository. None of the write operations defined by CORBA are supported, nor are set operations on nonread-only attributes.

Read access to the Interface Repository database is always through the Interface Repository server; that is, a client reads from the database by invoking methods that are performed by the server. The read operations as defined by the *CORBA Common Object Request Broker:*Architecture and Specification, Revision 2.4, are described in this chapter.

- Programming Information
- Performance Implications

### 10.1.1 Programming Information

The interface to a server is defined in the OMG IDL file. How the OMG IDL file is accessed depends on the type of client being built. Three types of clients are considered: stub based, Dynamic Invocation Interface (DII).



Client applications that use stub-style invocations need the OMG IDL file at build time. The programmer can use the OMG IDL file to generate stubs, and so forth. (For more information, see *Creating CORBA Client Applications*.) No other access to the Interface Repository is required.

Client applications that use the Dynamic Invocation Interface (DII) need to access the Interface Repository programmatically. The interface to the Interface Repository is defined in this chapter and is discussed in Building Client Applications. The exact steps taken to access the Interface Repository depend on whether the client is seeking information about a specific object, or browsing the Interface Repository to find an interface. To obtain information about a specific object, clients use the CORBA::Object::\_get\_interface method to obtain an InterfaceDef object. (Refer to CORBA::Object::\_get\_interface for a description of this method.) Using the InterfaceDef object, the client can get complete information about the interface.

Before a DII client can browse the Interface Repository, it needs to obtain the object reference of the Interface Repository to start the search.

DII clients use the Bootstrap object to obtain the object reference. (For a description of this method, see the section Tobj\_Bootstrap::register\_callback\_port.) Once the client has the object reference, it can navigate the Interface Repository, starting at the root.

To obtain a reference to a Interface Repository in the domain to which a client application is associated, the client application can use either of two bootstrapping mechanisms:

- Invoke the Tobj\_Bootstrap::resolve\_initial\_references operation with a value of "CORBA::Repository". This operation returns a reference to a InterfaceRepository object that is in the domain to which the client application is currently attached. You should use this mechanism if you are using the Oracle Tuxedo client software. For more information, see the section Tobj\_Bootstrap::resolve\_initial\_references.
- Invoke the CORBA::ORB::resolve\_initial\_references operation with a value of "CORBA::Repository". This operation returns a reference to a InterfaceRepository object that is in the domain to which the client application is currently attached. You should use this mechanism if you are using a third-party client ORB. For more information, see the section CORBA::ORB::resolve\_initial\_references.



To use the DII, the OMG IDL file must be stored in the Interface Repository.

### 10.1.2 Performance Implications

All run-time access to the Interface Repository is via the Interface Repository server. Because there is considerable overhead in making requests of a remote server application, designers need to be aware of this. For example, consider the interaction required to use an object reference to obtain the necessary information to make a DII invocation on the object reference. The steps are as follows:

- 1. The client application invokes the \_get\_interface operation on the CORBA::Object to get the InterfaceDef object associated with the object in question. This causes a message to be sent to the ORB that created the object reference.
- The ORB returns the InterfaceDef object to the client.
- The client invokes one or more \_is\_a operations on the object to determine what type of interface is supported by the object.



- 4. After the client has identified the interface, it invokes the describe\_interface operation on the Interface object to get a full description of the interface (for example, version number, operations, attributes, and parameters). This causes a message to be sent to the Interface Repository, and a reply is returned.
- The client is now ready to construct a DII request.

# 10.2 Building Client Applications

Clients that use the Interface Repository need to link in Interface Repository stubs. How this happens is specific to the vendor. If the client application is using the Oracle Tuxedo ORB, the Oracle Tuxedo software provides the stubs in the form of a library. Therefore, programmers do not need to use the Interface Repository OMG IDL file to build the stubs. The Interface Repository definitions are contained within the CORBA.h file, but they are not included by default.



To use the Interface Repository definitions, you must define the ORB\_INCLUDE\_REPOSITORY macro before including CORBA.h in your client application code (for example: #Define ORB INCLUDE REPOSITORY).

If the client application is using a third-party ORB (for example, ORBIX) the programmer must use the mechanisms that are provided by that vendor. This might include generating stubs from the OMG IDL file using the IDL compiler supplied by the vendor, simply linking against the stubs provided by the vendor, or some other mechanism.

Some third-party ORBs provide a local Interface Repository capability. In this case, the local Interface Repository is provided by the vendor and is populated with the interface definitions that are needed by that client.

# 10.3 Getting Initial References to the InterfaceRepository Object

You use the Bootstrap object to get an initial reference to the InterfaceRepository object. For a description of the Bootstrap object method, see the command <a href="Tobj">Tobj</a> Bootstrap::resolve initial references.

# 10.4 Interface Repository Interfaces

Client applications use the interfaces defined by CORBA to access the Interface Repository. This section contains descriptions of each interface that is implemented in the Oracle Tuxedo software.



The Oracle Tuxedo CORBA implementation of the Interface Repository only supports the read operations on the interfaces. The write operations are not implemented.

- Supporting Type Definitions
- IRObject Interface



- Contained Interface
- Container Interface
- IDLType Interface
- Repository Interface
- ModuleDef Interface
- ConstantDef Interface
- TypedefDef Interface
- StructDef
- UnionDef
- EnumDef
- AliasDef
- PrimitiveDef
- StringDef
- WstringDef
- ExceptionDef
- AttributeDef
- OperationDef
- InterfaceDef

### 10.4.1 Supporting Type Definitions

Several types are used throughout the Interface Repository interface definitions.

```
module CORBA {
                                       Identifier;
  typedef string
  typedef string
                                       ScopedName;
   typedef string
                                       RepositoryId;
   enum DefinitionKind {
     dk none, dk all,
     dk Attribute, dk Constant, dk Exception, dk Interface,
     dk Module, dk Operation, dk Typedef,
     dk Alias, dk Struct, dk Union, dk Enum,
     dk Primitive, dk String, dk Sequence, dk Array,
     dk Repository,
     dk Wstring, dk Fixed,
     dk Value, dk ValueBox, dk ValueMember,
     dk Native
     };
};
```

Identifiers are the simple names that identify modules, interfaces, value types, value members, value boxes, constants, typedefs, exceptions, attributes, operations, and native types. They correspond exactly to OMG IDL identifiers. An Identifier is not necessarily unique within an entire Interface Repository; it is unique only within a particular Repository, ModuleDef, InterfaceDef, ValueDef, or OperationDef.

A ScopedName is a name made up of one or more identifiers separated by double colons (::). They correspond to OMG IDL scoped names. An absolute ScopedName is one that begins with double colons (::) and unambiguously identifies a definition in a Repository. An absolute ScopedName in a Repository corresponds to a global name in an OMG IDL file. A relative ScopedName does not begin with double colons (::) and must be resolved relative to some context.

A RepositoryId is an identifier used to uniquely and globally identify a module, interface, value type, value member, value box, native type, constant, typedef, exception, attribute, or operation. Because RepositoryIds are defined as strings, they can be manipulated (for example, copied and compared) using a language binding's string manipulation routines.

A DefinitionKind identifies the type of an Interface Repository object.

### 10.4.2 IRObject Interface

The base interface IRObject (shown below) represents the most generic interface from which all other Interface Repository interfaces are derived, even the Repository itself.

```
module CORBA {
    interface IRObject {
        readonly attribute DefinitionKind def_kind;
    };
};
```

The def kind attribute identifies the type of the definition.

#### 10.4.3 Contained Interface

The Contained interface (shown below) is inherited by all Interface Repository interfaces that are contained by other Interface Repository objects. All objects within the Interface Repository, except the root object (Repository) and definitions of anonymous (ArrayDef, StringDef, and SequenceDef), and primitive types are contained by other objects.

```
module CORBA {
  typedef string VersionSpec;
  interface Contained : IRObject {
      readonly attribute RepositoryId
      readonly attribute Identifier
                                             name:
      readonly attribute VersionSpec
                                             version;
      readonly attribute Container
                                             defined in;
      readonly attribute ScopedName
                                             absolute name;
      readonly attribute Repository
                                             containing repository;
      struct Description {
         DefinitionKind
                                     kind;
                                     value;
         any
        };
      Description describe ();
      };
};
```

An object that is contained by another object has an id attribute that identifies it globally, and a name attribute that identifies it uniquely within the enclosing Container object. It also has a

version attribute that distinguishes it from other versioned objects with the same name. The Oracle Tuxedo CORBA Interface Repository does not support simultaneous containment or multiple versions of the same named object.

Contained objects also have a <code>defined\_in</code> attribute that identifies the Container within which they are defined. Objects can be contained either because they are defined within the containing object (for example, an interface is defined within a module) or because they are inherited by the containing object (for example, an operation may be contained by an interface because the interface inherits the operation from another interface). If an object is contained through inheritance, the <code>defined\_in</code> attribute identifies the InterfaceDef or ValueDef from which the object is inherited.

The absolute\_name attribute is an absolute <code>ScopedName</code> that identifies a Contained object uniquely within its enclosing Repository. If this object's <code>defined\_in</code> attribute references a Repository, the <code>absolute\_name</code> is formed by concatenating the string "::" and this object's <code>name</code> attribute. Otherwise, the <code>absolute\_name</code> is formed by concatenating the <code>absolute\_name</code> attribute of the object referenced by this object's <code>defined\_in</code> attribute, the string "::", and this object's <code>name</code> attribute.

The containing\_repository attribute identifies the Repository that is eventually reached by recursively following the object's defined in attribute.

The within operation returns the list of objects that contain the object. If the object is an interface or module, it can be contained only by the object that defines it. Other objects can be contained by the objects that define them and by the objects that inherit them.

The describe operation returns a structure containing information about the interface. The description structure associated with each interface is provided below with the interface's definition. The kind of definition described by the structure returned is provided with the returned structure. For example, if the describe operation is invoked on an attribute object, the kind field contains dk\_Attribute and the value field contains an any, which contains the AttributeDescription structure.

### 10.4.4 Container Interface

The base interface Container is used to form a containment hierarchy in the Interface Repository. A Container can contain any number of objects derived from the Contained interface. All Containers, except for Repository, are also derived from Contained.

```
module CORBA {
   typedef sequence <Contained> ContainedSeq;
   interface Container : IRObject {
     Contained lookup (in ScopedName search name);
     ContainedSeq contents (
          in DefinitionKind
                                   limit type,
          in boolean
                                   exclude inherited
     ContainedSeq lookup name (
          in Identifier
                                   search name,
          in long
                                   levels to search,
          in DefinitionKind
                                   limit type,
                                   exclude inherited
          in boolean
         );
  struct Description {
     Contained
                                   contained object;
```



```
DefinitionKind kind;
any value;
};
typedef sequence<Description> DescriptionSeq;
DescriptionSeq describe_contents (
    in DefinitionKind limit_type,
    in boolean exclude_inherited,
    in long max_returned_objs
    );
};
```

The lookup operation locates a definition relative to this container, given a scoped name using the OMG IDL rules for name scoping. An absolute scoped name (beginning double colons (::)) locates the definition relative to the enclosing Repository. If no object is found, a nil object reference is returned.

The contents operation returns the list of objects directly contained by or inherited into the object. The operation is used to navigate through the hierarchy of objects. Starting with the Repository object, a client uses this operation to list all of the objects contained by the Repository, all of the objects contained by the modules within the Repository, all of the interfaces and value types within a specific module, and so on.

#### limit type

If <code>limit\_type</code> is set to <code>dk\_all</code>, objects of all types are returned. For example, if this is an InterfaceDef, the attribute, operation, and exception objects are all returned. If <code>limit\_type</code> is set to a specific interface, only objects of that type are returned. For example, only attribute objects are returned if <code>limit\_type</code> is set to <code>dk\_Attribute</code>.

#### exclude inherited

If set to TRUE, inherited objects (if there are any) are not returned. If set to FALSE, all contained objects (whether contained due to inheritance or because they were defined within the object) are returned.

The <code>lookup\_name</code> operation is used to locate an object by name within a particular object or within the objects contained by that object. The <code>describe\_contents</code> operation combines the <code>contents</code> operation and the <code>describe</code> operation. For each object returned by the contents operation, the description of the object is returned (that is, the object's <code>describe</code> operation is invoked and the results are returned).

The <code>lookup\_name</code> operation is used to locate an object by name within a particular object or within the objects contained by that object.

#### search\_name

Specifies which name is to be searched for.

#### levels to search

Controls whether the lookup is constrained to the object the operation is invoked on, or whether the lookup should search through objects contained by the object as well. Setting levels\_to\_search to -1 searches the current object and all contained objects. Setting levels\_to\_search to 1 searches only the current object. Use of values of levels\_to\_search of 0 or of negative numbers other than -1 is undefined.



The describe\_contents operation combines the contents operation and the describe operation. For each object returned by the contents operation, the description of the object is returned (i.e., the object's describe operation is invoked and the results returned).

#### max returned objs

Limits the number of objects that can be returned in an invocation of the call to the number provided. Setting the parameter to -1 indicates return all contained objects.

# 10.4.5 IDLType Interface

The base interface IDLType (shown below) is inherited by all Interface Repository objects that represent OMG IDL types. It provides access to the TypeCode describing the type, and is used in defining other interfaces wherever definitions of IDL types must be referenced.

```
module CORBA {
    interface IDLType : IRObject {
        readonly attribute TypeCode type;
    };
};
```

The type attribute describes the type defined by an object derived from IDLType.

# 10.4.6 Repository Interface

Repository (shown below) is an interface that provides global access to the Interface Repository. The Repository object can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and modules. As it inherits from Container, it can be used to look up any definition (whether globally defined or defined within a module or an interface) either by name or by id.

Since the Repository derives only from Container and not from Contained, it does not have a Repositoryld associated with it. By default, it is deemed to have the Repositoryld ""(the empty string) for purposes of assigning a value to the defined\_in field of the description structure of ModuleDef, InterfaceDef, ValueDef, ValueBoxDef, TypedefDef, ExceptionDef, and ConstantDef that are contained immediately in the Repository object.

```
module CORBA {
    interface Repository : Container {
        Contained lookup_id (in RepositoryId search_id);
        TypeCode get_canonical_typecode(in TypeCode tc);
        PrimitiveDef get_primitive (in PrimitiveKind kind);
    };
};
```

The lookup\_id operation is used to look up an object in a Repository, given its RepositoryId. If the Repository does not contain a definition for search id, a nil object reference is returned.

The <code>get\_canonical\_typecode</code> operation looks up the TypeCode in the Interface Repository and returns an equivalent TypeCode that includes all repository IDs, names, and member\_names. If the top level TypeCode does not contain a Repositoryld, such as array and sequence TypeCodes, or TypeCodes from older ORBs, or if it contains a Repositoryld that is not found in the target Repository, then a new TypeCode is constructed by recursively calling <code>get\_canonical\_typecode</code> on each member TypeCode of the original TypeCode.

The get\_primitive operation returns a reference to a PrimitiveDef with the specified kind attribute. All PrimitiveDefs are immutable and are owned by the Repository.

### 10.4.7 ModuleDef Interface

A ModuleDef (shown below) can contain constants, typedefs, exceptions, interfaces, value types, value boxes, native types, and other module objects.

```
module CORBA {
    interface ModuleDef : Container, Contained {
    };
    struct ModuleDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
    };
};
```

The inherited describe operation for a ModuleDef object returns a ModuleDescription.

### 10.4.8 ConstantDef Interface

A ConstantDef object (shown below) defines a named constant.

```
module CORBA {
      interface ConstantDef : Contained {
            readonly attribute TypeCode
                                           type;
            readonly attribute IDLType
                                           type def;
            readonly attribute any
                                             value:
      };
      struct ConstantDescription {
             Identifier name;
             RepositoryId id;
RepositoryId defined_in;
             VersionSpec
                                version;
             TypeCode
                                 type;
             any
                                 value;
     };
};
```

#### type

Specifies the TypeCode describing the type of the constant. The type of a constant must be one of the simple types (long, short, float, char, string, octet, and so on).

#### type\_def

Identifies the definition of the type of the constant.

#### value

Contains the value of the constant, not the computation of the value (for example, the fact that it was defined as "1+2").

The describe operation for a ConstantDef object returns a ConstantDescription.

# 10.4.9 TypedefDef Interface

A TypedefDef (shown below) is an abstract interface used as a base interface for all named nonobject types (structures, unions, enumerations, and aliases). The TypedefDef interface is not inherited by the definition objects for primitive or anonymous types.

```
module CORBA {
    interface TypedefDef : Contained, IDLType {
    };
    struct TypeDescription {
        Identifier name;
        RepositoryId id;
        RepositoryId defined_in;
        VersionSpec version;
        TypeCode type;
    };
};
```

The inherited describe operation for interfaces derived from TypedefDef returns a TypeDescription.

### 10.4.10 StructDef

A StructDef (shown below) represents an OMG IDL structure definition. It contains the members of the struct.

```
module CORBA {
    struct StructMember {
        Identifier name;
        TypeCode type;
        IDLType type_def;
    };
    typedef sequence <StructMember> StructMemberSeq;
    interface StructDef : TypedefDef, Container{
            readonly attribute StructMemberSeq members;
    };
};
```

The members attribute contains a description of each structure member.

The inherited type attribute is a tk struct TypeCode describing the structure.

# 10.4.11 UnionDef

A UnionDef (shown below) represents an OMG IDL union definition. It contains the members of the union.

```
module CORBA {
    struct UnionMember {
        Identifier name;
        any label;
        TypeCode type;
```



```
IDLType type_def;
};
typedef sequence <UnionMember> UnionMemberSeq;

interface UnionDef : TypedefDef, Container {
    readonly attribute TypeCode discriminator_type;
    readonly attribute IDLType discriminator_type_def;
    readonly attribute UnionMemberSeq members;
};
};
```

#### discriminator type and discriminator type def

Describes and identifies the union's discriminator type.

#### members

Contains a description of each union member. The label of each UnionMemberDescription is a distinct value of the discriminator\_type. Adjacent members can have the same name. Members with the same name must also have the same type. A label with type octet and value 0 (zero) indicates the default union member.

The inherited type attribute is a tk union TypeCode describing the union.

### 10.4.12 EnumDef

An EnumDef (shown below) represents an OMG IDL enumeration definition.

#### members

Contains a distinct name for each possible value of the enumeration.

The inherited type attribute is a tk enum TypeCode describing the enumeration.

# 10.4.13 AliasDef

An AliasDef (shown below) represents an OMG IDL typedef that aliases another definition.

```
module CORBA {
    interface AliasDef : TypedefDef {
        readonly attribute IDLType original_type_def;
    };
};
```

#### original\_type\_def

Identifies the type being aliased.

The inherited type attribute is a tk alias TypeCode describing the alias.

### 10.4.14 PrimitiveDef

A PrimitiveDef (shown below) represents one of the OMG IDL primitive types. Because primitive types are unnamed, this interface is not derived from TypedefDef or Contained.

```
module CORBA {
   enum PrimitiveKind {
     pk_null, pk_void, pk_short, pk_long, pk_ushort, pk_ulong,
     pk_float, pk_double, pk_boolean, pk_char, pk_octet,
     pk_any, pk_TypeCode, pk_Principal, pk_string, pk_objref,
          pk_longlong, pk_ulonglong, pk_longdouble, pk_wchar, pk_wstring,
     pk_value_base
   };
interface PrimitiveDef: IDLType {
     readonly attribute PrimitiveKind kind;
   };
};
```

#### kind

Indicates which primitive type the PrimitiveDef represents. There are no PrimitiveDefs with kind  $pk\_null$ . A PrimitiveDef with kind  $pk\_string$  represents an unbounded string. A PrimitiveDef with kind  $pk\_objref$  represents the OMG IDL type Object. A PrimitiveDef with kind  $pk\_objref$  represents the IDL type ValueBase.

The inherited type attribute describes the primitive type.

All PrimitiveDefs are owned by the Repository. References to them are obtained using Repository::get primitive.

# 10.4.15 StringDef

A StringDef represents an IDL bounded string type. The unbounded string type is represented as a PrimitiveDef. As string types are anonymous, this interface is not derived from TypedefDef or Contained.

```
module CORBA {
   interface StringDef : IDLType {
      attribute unsigned long bound;
   };
};
```

The bound attribute specifies the maximum number of characters in the string and must not be zero.

The inherited type attribute is a tk\_string TypeCode describing the string.

# 10.4.16 WstringDef

A WstringDef represents an IDL wide string. The unbounded wide string type is represented as a PrimitiveDef. As wide string types are anonymous, this interface is not derived from TypedefDef or Contained.

```
module CORBA {
   interface WstringDef : IDLType {
     attribute unsigned long bound;
   };
};
```

The bound attribute specifies the maximum number of wide characters in a wide string, and must not be zero.

The inherited type attribute is a tk wstring TypeCode describing the wide string.

# 10.4.17 ExceptionDef

An ExceptionDef (shown below) represents an exception definition. It can contain structs, unions, and enums.

```
module CORBA {
    interface ExceptionDef : Contained, Container {
         readonly attribute TypeCode
                                               type;
         readonly attribute StructMemberSeq members;
   };
struct ExceptionDescription {
     Identifier
                            name;
     RepositoryId
                            id;
     RepositoryId
                           defined in;
     VersionSpec
                            version;
     TypeCode
                            type;
  } ;
```

#### type

tk except TypeCode that describes the exception.

#### members

Describes any exception members.

The describe operation for a ExceptionDef object returns an ExceptionDescription.

# 10.4.18 AttributeDef

An AttributeDef (shown below) represents the information that defines an attribute of an interface.

```
module CORBA {
    enum AttributeMode {ATTR_NORMAL, ATTR_READONLY};
    interface AttributeDef : Contained {
```

```
readonly attribute TypeCode type;
    attribute IDLType type_def;
    attribute AttributeMode mode;
};
struct AttributeDescription {
    Identifier name;
    RepositoryId id;
    RepositoryId defined_in;
    VersionSpec version;
    TypeCode type;
    AttributeMode mode;
};
};
```

#### type

Provides the TypeCode describing the type of this attribute.

#### type def

Identifies the object that defines the type of this attribute.

#### mode

Specifies read only or read/write access for this attribute.

The describe operation for an AttributeDef object returns an AttributeDescription.

# 10.4.19 OperationDef

An OperationDef (shown below) represents the information needed to define an operation of an interface.

```
module CORBA {
     enum OperationMode {OP NORMAL, OP ONEWAY};
     enum ParameterMode {PARAM IN, PARAM OUT, PARAM INOUT};
     struct ParameterDescription {
           Identifier
                               name;
           TypeCode
                                type;
           IDLType
                                type def;
           ParameterMode
                               mode;
     typedef sequence <ParameterDescription> ParDescriptionSeq;
     typedef Identifier ContextIdentifier;
     typedef sequence <ContextIdentifier> ContextIdSeq;
     typedef sequence <ExceptionDef> ExceptionDefSeq;
     typedef sequence <ExceptionDescription> ExcDescriptionSeq;
     interface OperationDef : Contained {
         readonly attribute TypeCode
                                                     result;
         readonly attribute IDLType readonly attribute ParDescriptionSeq params;
                                                     result def;
         readonly attribute OperationMode
                                                     mode;
         readonly attribute ContextIdSeq
                                                    contexts;
         readonly attribute ExceptionDefSeq
                                                      exceptions;
    };
```

```
struct OperationDescription {
    Identifier
                                     name;
    RepositoryId
                                     id;
    RepositoryId
                                     defined in;
                                     version;
   VersionSpec
   TypeCode
                                     result;
    OperationMode
                                    mode;
    ContextIdSeq
                                    contexts;
   ParDescriptionSeq
                                    parameters;
    ExcDescriptionSeq
                                    exceptions;
 };
};
```

#### result

A TypeCode that describes the type of the value returned by the operation.

#### result def

Identifies the definition of the returned type.

#### params

Describes the parameters of the operation. It is a sequence of ParameterDescription structures. The order of the ParameterDescriptions in the sequence is significant. The <code>name</code> member of each structure provides the parameter name. The <code>type</code> member is a TypeCode describing the type of the parameter. The <code>type\_def</code> member identifies the definition of the type of the parameter. The <code>mode</code> member indicates whether the parameter is an in, out, or inout parameter.

#### mode

The operation's mode is either oneway (that is, no output is returned) or normal.

#### contexts

Specifies the list of context identifiers that apply to the operation.

#### exceptions

Specifies the list of exception types that can be raised by the operation.

The inherited describe operation for an OperationDef object returns an OperationDescription.

The inherited <code>describe\_contents</code> operation provides a complete description of this operation, including a description of each parameter defined for this operation.

# 10.4.20 InterfaceDef

An InterfaceDef object (shown below) represents an interface definition. It can contain constants, typedefs, exceptions, operations, and attributes.

```
module CORBA {
  interface InterfaceDef;
    typedef sequence <InterfaceDef> InterfaceDefSeq;
    typedef sequence <RepositoryId> RepositoryIdSeq;
    typedef sequence <OperationDescription> OpDescriptionSeq;
    typedef sequence <AttributeDescription> AttrDescriptionSeq;

interface InterfaceDef: Container, Contained, IDLType {
        readonly attribute InterfaceDefSeq base_interfaces;
        readonly attribute boolean is_abstract;
```

```
boolean is a (in RepositoryId interface id);
     struct FullInterfaceDescription {
         Identifier
         RepositoryId
RepositoryId
VersionSpec
                                id;
                            defined_in;
                               version;
         OpDescriptionSeq operations;
         AttrDescriptionSeq
                                attributes;
         RepositoryIdSeq base_interfaces;
         TypeCode
                                type;
         boolean
                                is abstract;
    };
     FullInterfaceDescription describe interface();
};
struct InterfaceDescription {
        Identifier
                                name;
        RepositoryId id;
RepositoryId defined_in;
VersionSpec version;
                              base_interfaces;
is_abstract;
        RepositoryIdSeq
        boolean
    } ;
};
```

The base interfaces attribute lists all the interfaces from which this interface inherits.

The is abstract attribute is TRUE if the interface is an abstract interface type.

The  $is_a$  operation returns <code>TRUE</code> if the interface on which it is invoked either is identical to or inherits, directly or indirectly, from the interface identified by its <code>interface\_id</code> parameter. Otherwise, it returns <code>FALSE</code>.

The describe\_interface operation returns a FullInterfaceDescription describing the interface, including its operations and attributes. The operations and attributes fields of the FullInterfaceDescription structure include descriptions of all of the operations and attributes in the transitive closure of the inheritance graph of the interface being described.

The inherited describe operation for an InterfaceDef returns an InterfaceDescription.

The inherited contents operation returns the list of constants, typedefs, and exceptions defined in this InterfaceDef and the list of attributes and operations either defined or inherited in this InterfaceDef. If the <code>exclude\_inherited</code> parameter is set to <code>TRUE</code>, only attributes and operations defined within this interface are returned. If the <code>exclude\_inherited</code> parameter is set to <code>FALSE</code>, all attributes and operations are returned.

11

# Joint Client/Servers

This chapter describes programming requirements for CORBA joint client/servers and the C++ OracleWrapper Callbacks API.



The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, etc. should only be used:

- · to help implement/run third party Java ORB libraries, and
- for programmer reference only.
   Technical support for third party CORBA Java ORBs should be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

This topic includes the following sections:

- Introduction. This section describes:
  - Main Program and Server Initialization
  - Servants
  - Servant Inheritance from Skeletons
  - Callback Object Models Supported
  - Configuring Servers to Call Remote Joint Client/Server Objects
  - Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only)
  - Preparing Callback Objects Using OracleWrapper Callbacks
- C++ OracleWrapper Callbacks Interface API
- Introduction
- C++ OracleWrapper Callbacks Interface API

# 11.1 Introduction

For either an Oracle Tuxedo CORBA client or joint client/server (that is, a client that can receive and process object invocations), the programmer writes the client main(). The main() uses Oracle Tuxedo CORBA environmental objects to establish connections, set up security, and start transactions.

Oracle Tuxedo clients invoke operations on objects. In the case of DII, client code creates the DII Request object and then invokes one of two operations on the DII Request. In the case of static invocation, client code performs the invocation by performing what looks like an ordinary invocation (which ends up calling code in the generated client stub). Additionally, the client

programmer uses ORB interfaces defined by OMG, and Oracle Tuxedo CORBA environmental objects that are supplied with the Oracle Tuxedo software, to perform functions unique to Oracle Tuxedo.

For Oracle Tuxedo joint client/server applications, the client code must be structured so that it can act as a server for callback Oracle Tuxedo objects. Such clients do not use the TP Framework and are not subject to Oracle Tuxedo system administration. Besides the programming implications, this means that CORBA joint client/servers do not have the same scalability and reliability as Oracle Tuxedo CORBA servers, nor do they have the state management and transaction behavior available in the TP Framework. If a user wants to have those characteristics, the application must be structured in such a way that the object implementations are in an Oracle Tuxedo CORBA server, rather than in a client.

The following sections describe the mechanisms you use to add callback support to an Oracle Tuxedo client. In some cases, the mechanisms are contrasted with the Oracle Tuxedo server mechanisms that use the TP Framework.

- Main Program and Server Initialization
- Servants
- Servant Inheritance from Skeletons
- Callback Object Models Supported
- Configuring Servers to Call Remote Joint Client/Server Objects
- Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only)
- Preparing Callback Objects Using OracleWrapper Callbacks

# 11.1.1 Main Program and Server Initialization

In an Oracle Tuxedo server, you use the buildobjserver command to create the main program for the C++ server. (Java servers are not supported in release 8.0 and later of Oracle Tuxedo.) Server main program takes care of all Oracle Tuxedo- and CORBA-related initialization of the server functions. However, since you implement the Server object, you have an opportunity to customize the way in which the server application is initialized and shut down. The server main program automatically invokes methods on the Server object at the appropriate times.

In contrast, for an Oracle Tuxedo CORBA joint client/server (as for an Oracle Tuxedo CORBA client), you create the main program and are responsible for all initialization. You do not need to provide a Server object because you have complete control over the main program and you can provide initialization and shutdown code in any way that is convenient.

The specific initialization needed for a joint client/server is discussed in the section Servants.

# 11.1.2 Servants

Servants (method code) for joint client/servers are very similar to servants for servers. All business logic is written the same way. The differences result from not using the TP Framework. Therefore, the main difference is that you use CORBA functions directly instead of indirectly through the TP Framework.

The Server interface is used in Oracle Tuxedo CORBA servers to allow the TP Framework to ask the user to create a servant for an object when the ORB receives a request for that object. However, in joint client/servers, the user program is responsible for creating a servant before any requests arrive; thus, the Server interface is not needed. Typically, the program creates a servant and then activates the object (using the servant and an ObjectId; the ObjectId is



possibly system generated) before handing a reference to the object. Such an object might be used to handle callbacks. Thus, the servant already exists and the object is activated before a request for the object arrives.

For C++ joint client/servers, instead of invoking the TP interface to perform certain operations, client servants directly invoke the ORB and POA (which is what the TP interface does internally). Alternately, since much of the interaction with the ORB and POA is the same for all applications, for ease of use, the client library provides a convenience wrapper object that does the same things, using a single operation. For a discussion of how to use the convenience wrapper object, see Callback Object Models Supported and Preparing Callback Objects Using OracleWrapper Callbacks.

### 11.1.3 Servant Inheritance from Skeletons

In a client that supports callbacks, as well as in a server, you write a implementation class that inherits from the same skeleton class name generated by the IDL compiler (the idl command).

• C++ Example of Inheritance from Skeletons

# 11.1.3.1 C++ Example of Inheritance from Skeletons

The following is a C++ example, given the IDL:

```
interface Hospital{ ... };
```

The skeleton generated by the idl command contains a "skeleton" class, POA\_Hospital, that the user-written class inherits from, as in:

```
class Hospital i : public POA Hospital { ... };
```

In a server, the skeleton class inherits from the TP Framework class <code>Tobj\_ServantBase</code>, which in turn inherits from the predefined <code>PortableServer::ServantBase</code>.

The inheritance tree for a callback object implementation in a joint client/server is different than that in a server. The skeleton class does not inherit from the TP Framework class

Tobj\_ServantBase, but instead inherits directly from PortableServer::ServantBase. This behavior is achieved by specifying the -P option in the idl command.

Not having the <code>Tobj\_ServantBase</code> class in the inheritance tree for a servant means that the servant does not have <code>activate\_object</code> and <code>deactivate\_object</code> methods. In a server, these methods are called by the TP Framework to dynamically initialize and save a servant's state before invoking a method on the servant. For a client that supports callbacks, you must write code that explicitly creates a servant and initializes a servant's state.

# 11.1.4 Callback Object Models Supported

Oracle Tuxedo CORBA supports four kinds of callback objects and provides wrappers for the three that are most common. These objects correspond to three combinations of POA policies. The POA policies control both the types of objects and the types of object references that are possible.

The POA policies that are applicable are:

- The LifeSpanPolicy, which controls how long an object reference is valid.
- The IdAssignmentPolicy, which controls who assigns the <code>ObjectId</code>—the user or the system.



These objects are explained primarily in terms of their behavioral characteristics rather than in details about how the ORB and the POA handle them. Those details are discussed in the next sections, using either direct ORB and POA calls (which requires a little extra knowledge of CORBA servers) or using the OracleWrapper Callbacks interface, which hides the ORB and POA calls (for users who do not care about the details).

- Transient/SystemId—object references are valid only for the life of the client process. The ObjectId is not assigned by the client application, but is a unique value assigned by the system. This type of object is useful for invocations that a client wants to receive only until the client terminates. (The corresponding POA LifeSpanPolicy value is TRANSIENT and the IdAssignmentPolicy is SYSTEM ID.)
- Persistent/SystemId —object references are valid across multiple activations. The
   ObjectId is not assigned by the client application, but is a unique value assigned by the
   system. This type of object and object reference is useful when the client goes up and
   down over a period of time. When the client is up, it can receive callback objects on that
   particular object reference.
  - Typically, the client will create the object reference once, save it in its own permanent storage area, and reactivate the servant for that object every time it comes up. If used with an Oracle Tuxedo CORBA Notification Service application, for example, these are callbacks that correspond to the concept of a persistent subscription; that is, the Notification Service remembers the callback reference and delivers events any time the client is up and declares that it is again ready to receive events. This allows notification service subscriptions to survive client failures or offline-time. (The corresponding POA policy values are PERSISTENT and SYSTEM ID.)
- Persistent/UserId—this is the same as Persistent/SystemId with the exception that the ObjectId has to be assigned by the client application. Such an ObjectId might be, for example, a database key meaningful only to the client. (The corresponding POA policy values are PERSISTENT and USER ID.)

#### Note:

The Transient/UserId policy combination is not considered particularly important. It is possible for users to provide for themselves by using the POA in a manner analogous to either of the persistent cases, but the Oracle Tuxedo wrappers do not provide special help to do so. For Oracle Tuxedo CORBA native joint client/servers, neither of the Persistent policies is supported, only the Transient policy.

# 11.1.5 Configuring Servers to Call Remote Joint Client/Server Objects

In order for an Oracle Tuxedo server to call remote joint client/server objects, that is, joint client/server objects located outside the Oracle Tuxedo domain, the server must be configured to enable outbound IIOP. This capability is enabled by specifying the -0 (uppercase letter O) option in the IIOP Server Listener (ISL) server command. Setting the -0 option enables outbound invokes (outbound IIOP) on joint client/server objects that are not connected to an IIOP Listener Handler (ISH).

You set ISL command options in the SERVERS section of the server's UBBCONFIG file. Because support for outbound IIOP requires a small amount of extra resources, the default is outbound IIOP disabled. For more information, see "Using the ISL Command to Configure Outbound IIOP" in Oracle® Tuxedo Application Configuration Guide and "ISL(1)" in the Oracle® Tuxedo Command Reference.



# 11.1.6 Preparing Callback Objects Using CORBA (C++ Joint Client/Servers Only)

To set up Oracle Tuxedo C++ callback objects using CORBA, the client must do the following:

- Establish a connection with a POA with the appropriate policies for the callback object model. (This can be the root POA, available by default, or it may require creating a new POA.)
- 2. Create a servant (that is, an instance of the C++ implementation class for the interface).
- 3. Inform the POA that the servant is ready to accept requests on the callback object. Technically, this means the client activates the object in the POA (that is, puts the servant and the ObjectId into the POA's Active Object Map).
- 4. Tell the POA to start accepting requests from the network (that is, activate the POA itself).
- 5. Create an object reference for the callback object.
- 6. Give out the object reference. This usually happens by making an invocation on another object with the callback object reference as a parameter (that is, the parameter is a callback object). That other object will then invoke the callback object (perform a callback invocation) at some later time.

Assuming that the client already has obtained a reference to the ORB, performing this task takes four interactions with the ORB and the POA. It might look like the model show in the following code snippet. In this model, only the Root POA is needed.

To use the Persistent/UserId model, there are some additional steps required when creating a POA. Further, the <code>ObjectId</code> is specified by the client, and this requires more steps. It might look like the model shown in the following code snippet.



```
4 CORBA::PolicyList policies(2);
5 policies.length(2);
6 policies[0] = root poa->create_lifespan_policy(
           PortableServer::PERSISTENT);
7 policies[1] = root poa->create id assignment policy(
           PortableServer:: USER ID );
8 PortableServer::POA var my poa ref =
           root poa->create POA(
"my poa ref", root poa->the POAManager(), policies);
9 root poa->the POAmanager()->activate();
// Create object reference for callback Object
10 oref = my poa ref -> create reference with id(
                                         oid, _tc_Catcher->id());
11 Catcher var my catcher ref = Catcher:: narrow( oref );
// activate object
12 my poa ref -> activate object with id( oid, my catcher i );
// Make the call passing the callback ref
    foo -> register callback ( my catcher ref );
```

All the interfaces and operations described here are standard CORBA interfaces and operations.

# 11.1.7 Preparing Callback Objects Using OracleWrapper Callbacks

You can use the OracleWrapper callbacks API with to write either C++ joint client/servers.

Using OracleWrapper Callbacks With C++

# 11.1.7.1 Using OracleWrapper Callbacks With C++

Because the code required for callback objects is nearly identical for every client that supports callbacks, you may find it convenient to use the OracleWrappers provided in the library provided for joint client/servers.

The OracleWrappers are described in IDL, as shown in the following code snippet.



```
in CORBA::RepositoryId
                                                       rep id)
       raises (ServantAlreadyActive);
    // set up persistent/systemid callback Object
    Object start persistent systemid(
           in PortableServer::Servant
                                                      servant,
           in CORBA::Repository
                                                     rep id,
           out string
                                                      stroid)
      raises (ServantAlreadyActive);
// reinstate set up for persistent/systemid
// callback object
Object restart persistent systemid(
         in PortableServer::Servant
                                                     servant,
         in CORBA::RepositoryId
                                                    rep id,
         in string
                                                     stroid)
    raises (ServantAlreadyActive, ObjectAlreadyActive);
// set up persistent/userid callback Object
  Object start persistent userid(
        in PortableServer::Servant
                                                    servant,
        in CORBA::RepositoryId
                                                    rep id,
        in string
                                                    stroid)
  raises (ServantAlreadyActive, ObjectAlreadyActive);
// stop servicing a particular callback Object
// with the given servant
void stop object( in PortableServer::Servant servant);
//Stop all callback Object processing
void stop all objects();
// get oid string for the current request
string get string oid() raises (NotInRequest);
};
#endif /* BEA WRAPPER IDL */
```

The OracleWrappers are described in C++ as shown in the following code snippet.

```
#ifndef BEAWRAPPER H
#define BEAWRAPPER H
#include <PortableServer.h>
class BEAWrapper{
class Callbacks{
      public:
            Callbacks (CORBA::ORB ptr init orb);
            CORBA::Object ptr start transient (
                         PortableServer::Servant servant,
                         const char *
                                            rep id);
            CORBA::Object ptr start persistent systemid (
                         PortableServer::Servant servant,
                         const char *
                                            rep id,
                         char *
                                            & stroid);
           CORBA:: Object ptr restart persistent systemid (
                         PortableServer::Servant servant,
                         const char * rep_id,
const char * stroid);
          CORBA:: Object ptr start persistent userid (
                        PortableServer::Servant servant,
                        const char *
                                          rep id,
```

# 11.2 C++ OracleWrapper Callbacks Interface API

This C++ OracleWrapper Callbacks interface API is described in the following sections.

- Callbacks
- start transient
- start\_persistent\_systemid
- · restart\_persistent\_systemid
- start\_persistent\_userid
- stop\_object
- stop\_all\_objects
- get\_string\_oid
- ~Callbacks

# 11.2.1 Callbacks

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Value

# 11.2.1.1 Synopsis

Returns a reference to the Callbacks interface.

# 11.2.1.2 C++ Binding

```
BEAWrapper::Callbacks( CORBA::ORB ptr init orb);
```

# 11.2.1.3 Argument

init ork

The ORB to be used for all further operations.

# 11.2.1.4 Exception

CORBA::IMP\_LIMIT

The BEAWrapper::Callbacks class has already be instantiated with an ORB pointer. Only one instance of this class can be used in a process. Users who need additional flexibility should use the POA directly.

# 11.2.1.5 Description

The constructor returns a reference to the Callbacks interface. Only one such object should be created for the process, even if multiple threads are used. Using more than one such object will result in undefined behavior.

### 11.2.1.6 Return Value

A reference to the Callbacks object.

# 11.2.2 start transient

- Synopsis
- IDL
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 11.2.2.1 Synopsis

Activates an object, sets the ORB and the POA to the proper state, and returns an object reference to the activated object.

### 11.2.2.2 IDL

```
Object start_transient( in PortableServer::Servant servant,
in CORBA::RepositoryId rep_id)
raises ( ServantAlreadyActive );
```

# 11.2.2.3 C++ Binding

```
CORBA::Object_ptr start_transient(
PortableServer::Servant servant,
const char* rep id);
```

# 11.2.2.4 Arguments

#### servant

An instance of the C++ implementation class for the interface.

#### rep id

The repository id of the interface.

### 11.2.2.5 Exceptions

#### ServantAlreadyActive

The servant is already being used for a callback. A servant can be used only for a callback with a single <code>ObjectId</code>. To receive callbacks on objects containing different <code>ObjectIds</code>, you must create different servants and activate them separately. The same servant can be reused only if a <code>stop\_object</code> operation tells the system to stop using the servant for its original <code>ObjectId</code>.

#### CORBA::BAD PARAM

The repository ID was a NULL string or the servant was a NULL pointer.

# 11.2.2.6 Description

This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep\_id, using an ObjectId generated by the system.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the activated object. The returned object reference will be
  valid only until the termination of the client or until the callback servant is halted by the user
  via the stop\_object operation; after that, invocations on that object reference are no
  longer valid and can never be made valid.

### 11.2.2.7 Return Value

#### CORBA::Object\_ptr

A reference to the object that was created with the <code>ObjectId</code> generated by the system and the <code>rep\_id</code> provided by the user. The object reference will need to be converted to a specific object type by invoking the  $_{\tt narrow()}$  operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

# 11.2.3 start persistent systemid

- Synopsis
- IDL



- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 11.2.3.1 Synopsis

Activates an object, sets the ORB and the POA to the proper state, sets the output parameter stroid, and returns an object reference to the activated object.

### 11.2.3.2 IDL

```
Object start_persistent_systemid(
in PortableServer::Servant servant,
in CORBA::RepositoryId rep_id,
out string stroid)
raises ( ServantAlreadyActive );
```

# 11.2.3.3 C++ Binding

```
CORBA::Object_ptr start_persistent_systemid(
PortableServer::Servant servant,
const char* rep_id,
char*& stroid);
```

# 11.2.3.4 Arguments

#### servant

An instance of the C++ implementation class for the interface.

#### rep id

The repository ID of the interface.

#### stroid

This argument is set by the system and is opaque to the user. The client will use it when it reactivates the object at a later time (using restart\_persistent\_systemid), most likely after the client process has terminated and restarted.

# 11.2.3.5 Exceptions

#### ServantAlreadyActive

The servant is already being used for a callback. A servant can be used only for a callback with a single <code>ObjectId</code>. To receive callbacks on objects containing different <code>ObjectIds</code>, you must create different servants and activate them separately. The same servant can be reused only if a <code>stop</code> operation tells the system to stop using the servant for its original <code>ObjectId</code>.

#### CORBA::BAD PARAMETER

The repository ID was a NULL string or the servant was a NULL pointer.

#### CORBA::IMP\_LIMIT

In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

### 11.2.3.6 Description

This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep\_id, using an ObjectId generated by the system.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Sets the output parameter stroid to the stringified version of an ObjectId assigned by the system.
- Returns an object reference to the activated object. The returned object reference will be valid even after termination of the client. That is, if the client terminates, restarts again, and then activates a servant with the same rep\_id and for the same ObjectId, the servant will accept requests made on that same object reference. Since the ObjectId was generated by the system, the application has to save that ObjectId.

### 11.2.3.7 Return Value

#### CORBA::Object ptr

An object reference created with the <code>ObjectId</code> generated by the system and the <code>rep\_id</code> provided by the user. The object reference will need to be converted to a specific object type by invoking the <code>\_narrow()</code> operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

# 11.2.4 restart persistent systemid

- Synopsis
- IDL
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 11.2.4.1 Synopsis

Activates an object, sets the ORB and the POA to the proper state, and returns an object reference to the activated object.

### 11.2.4.2 IDL

```
Object restart_persistent_systemid(
in PortableServer::Servant servant,
in CORBA::RepositoryId rep id,
```

```
in string stroid)
raises (ServantAlreadyActive, ObjectAlreadyActive);
```

# 11.2.4.3 C++ Binding

```
CORBA::Object_ptr restart_persistent_systemid(
PortableServer::Servant servant,
const char* rep_id
const char* stroid);
```

# 11.2.4.4 Arguments

#### servant

An instance of the C++ implementation class for the interface.

#### rep id

The repository ID of the interface.

#### stroid

The stringified version of the <code>ObjectId</code> provided by the user to be set in the object reference being created. It must have been returned from a previous call on start persistent systemid.

# 11.2.4.5 Exceptions

#### ServantAlreadyActive

The servant is already being used for a callback. A servant can be used only for a callback with a single <code>ObjectId</code>. To receive callbacks on objects containing different <code>ObjectIds</code>, you must create different servants and activate them separately. The same servant can be reused only if a <code>stop\_object</code> operation tells the system to stop using the servant for its original <code>ObjectId</code>.

#### ObjectAlreadyActive

The stringified <code>ObjectId</code> is already being used for a callback. A given <code>ObjectId</code> can have only one servant associated with it. If you wish to change to a different servant, you must first invoke <code>stop</code> <code>object</code> with the servant currently in use.

#### CORBA::BAD PARAM

The repository ID was a NULL string or the servant was a NULL pointer or the ObjectId supplied was not previously assigned by the system.

#### CORBA::IMP LIMIT

In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

# 11.2.4.6 Description

This operation performs the following actions:

Activates an object using the Servant supplied to service objects of the type rep\_id, using
the supplied stroid (a stringified ObjectId), which must have been obtained by a previous
call on start persistent systemid.

- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the object activated.
- The reactivation would be done using the restart persistent systemid method.

#### 11.2.4.7 Return Value

#### CORBA::Object ptr

An object reference created with the stringified <code>ObjectId</code> stroid and the <code>rep\_id</code> provided by the user. The object reference will need to be converted to a specific object type by invoking the <code>\_narrow()</code> operation defined for the specific object. The caller is responsible for releasing the object when done.

# 11.2.5 start\_persistent\_userid

- Synopsis
- IDL
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Value

# 11.2.5.1 Synopsis

Activates an object, sets the ORB and the POA to the proper state, and returns an object reference to the activated object.

### 11.2.5.2 IDL

```
Object start_persistent_userid(
portableServer::Servant a_servant,
in CORBA::RepositoryId rep_id,
in string stroid)
raises ( ServantAlreadyActive, ObjectAlreadyActive );
```

# 11.2.5.3 C++ Binding

```
CORBA::Object_ptr start_persistent_userid (
PortableServer::Servant servant,
const char* rep_id,
const char* stroid);
```

# 11.2.5.4 Arguments

#### servant

An instance of the C++ implementation class for the interface.

#### rep id

The repository ID of the interface.

#### stroid

The stringified version of an ObjectId provided by the user to be set in the object reference being created. The stroid holds application-specific data and is opaque to the ORB.

### 11.2.5.5 Exceptions

#### ServantAlreadyActive

The servant is already being used for a callback. A servant can be used only for a callback with a single <code>ObjectId</code>. To receive callbacks on objects containing different <code>ObjectIds</code>, you must create different servants and activate them separately. The same servant can be reused only if a <code>stop\_object</code> operation tells the system to stop using the servant for its original <code>ObjectId</code>.

#### ObjectAlreadyActive

The stringified <code>ObjectId</code> is already being used for a callback. A given <code>ObjectId</code> can have only one servant associated with it. If you wish to change to a different servant, you must first invoke <code>stop</code> <code>object</code> with the servant currently in use.

#### CORBA::BAD PARAM

The repository ID was a NULL string or the servant was a NULL pointer.

#### CORBA::IMP LIMIT

In addition to other system reasons for this exception, a reason unique to this situation is that the joint client/server was not initialized with a port number; therefore, a persistent object reference cannot be created.

# 11.2.5.6 Description

This operation performs the following actions:

- Activates an object using the Servant supplied to service objects of the type rep\_id, using the object Id stroid.
- Sets the ORB and the POA into the state in which they will accept requests on that object.
- Returns an object reference to the activated object. The returned object reference will be valid even after termination of the client. That is, if the client terminates, and restarts again, and then activates a servant with the same rep\_id and for the same ObjectId, the servant will accept requests made on that same object reference.

# 11.2.5.7 Return Value

#### CORBA::Object\_ptr

An object reference created with the stringified <code>ObjectId</code> <code>stroid</code> and the <code>rep\_id</code> provided by the user. The object reference will need to be converted to a specific object type by invoking the <code>\_narrow()</code> operation defined for the specific object. The caller is responsible for releasing the object when the conversion is done.

# 11.2.6 stop\_object

- Synopsis
- IDL



- C++ Binding
- Argument
- Exceptions
- Description
- Return Value

# 11.2.6.1 Synopsis

Tells the ORB to stop accepting requests on the object that is using the given servant.

### 11.2.6.2 IDL

```
void stop object( in PortableServer::Servant servant);
```

# 11.2.6.3 C++ Binding

```
void stop_object(PortableServer::Servant servant);
```

### 11.2.6.4 Argument

#### servant

An instance of the C++ implementation class for the interface. The association between this servant and its <code>ObjectId</code> will be removed from the Active Object Map.

# 11.2.6.5 Exceptions

None.

# 11.2.6.6 Description

This operation tells the ORB to stop accepting requests on the given servant. It does not matter what state the servant is in, activated or deactivated; no error is reported.



If you do an invocation on a callback object after you call the <code>stop\_object</code> operation, the <code>OBJECT\_NOT\_EXIST</code> exception is returned to the caller. This is because the <code>stop object</code> operation, in effect, deletes the object.

### 11.2.6.7 Return Value

None.

# 11.2.7 stop\_all\_objects

- Synopsis
- IDL

- C++ Binding
- Exceptions
- Description
- Usage Note
- Return Value

# 11.2.7.1 Synopsis

Tells the ORB to stop accepting requests on all servants.

### 11.2.7.2 IDL

```
void stop all objects ();
```

# 11.2.7.3 C++ Binding

```
void stop all objects ();
```

### 11.2.7.4 Exceptions

None.

# 11.2.7.5 Description

This operation tells the ORB to stop accepting requests on all servants that have been set up in this process.

# 11.2.7.6 Usage Note

If a client calls the ORB::shutdown method, then it must not subsequently call stop\_all\_objects.

### 11.2.7.7 Return Value

None.

# 11.2.8 get\_string\_oid

- Synopsis
- IDL
- C++ Binding
- JExceptions
- Description
- Return Value

# 11.2.8.1 Synopsis

Requests the string version of the ObjectId of the current request.

### 11.2.8.2 IDL

```
string get string oid() raises (NotInRequest);
```

### 11.2.8.3 C++ Binding

```
char* get string oid();
```

# 11.2.8.4 JExceptions

#### NotInRequest

The function was called when the ORB was not in the context of a request (that is, not while the ORB was servicing a request in method code). Do not call this function from client code. It is legal only during the execution of a method of the callback object (that is, the servant).

### 11.2.8.5 Description

This operation returns the string version of the <code>ObjectId</code> of the current request.

### 11.2.8.6 Return Value

#### char\*

The string version of the <code>ObjectId</code> of the current request. This is the string that was supplied when the object reference was created. The string is meaningful to users only in the case when the object reference was created by the <code>start\_persistent\_userid</code> function. (That is, the <code>ObjectId</code> created by <code>start\_transient</code> and <code>start\_persistent\_systemid</code> were created by the ORB and has no relationship to the user application.)

# 11.2.9 ~Callbacks

- Synopsis
- · C++ Binding
- Arguments
- Exceptions
- Description
- Usage Note
- Return Value

# 11.2.9.1 Synopsis

Destroys the callback object.

# 11.2.9.2 C++ Binding

```
BEAWrapper::~Callbacks();
```

# 11.2.9.3 Arguments

None.

# 11.2.9.4 Exceptions

None.

# 11.2.9.5 Description

This destructor destroys the callback object.

# 11.2.9.6 Usage Note

If a client wants to get rid of the wrapper, but not shut down the ORB, then the client must call the  $stop\ all\ objects\ method.$ 

# 11.2.9.7 Return Value

None.



12

# **Development Commands**

For a detailed discussion of Oracle Tuxedo development commands, see the Oracle Tuxedo Command Reference. This document describes all Oracle Tuxedo commands and processes.

A PDF file of the Oracle Tuxedo Command Reference is also provided in the online documentation.



# Mapping of OMG IDL Statements to C++



Some of the information in this chapter is taken from the *Common Object Request Broker: C++ Language Mapping Specification*, June 1999, published by the Object Management Group (OMG). Used with permission of the OMG.

This chapter discusses the mappings from OMG IDL statements to C++.

- Mappings
- Fixed-length Versus Variable-length User-defined Types
- Using var Classes
- Using out Classes
- Argument Passing Considerations

# 13.1 Mappings

OMG IDL-to-C++ mappings are described for the following:

- Data Types
- Strings
- wchars
- wstrings
- Constants
- Enums
- Structs
- Unions
- Sequences
- Arrays
- Exceptions
- Mapping of Pseudo-objects to C++
- Usage
- Mapping Rules
- Relation to the C PIDL Mapping
- Typedefs
- Implementing Interfaces
- Implementing Operations

- PortableServer Functions
- Modules
- Interfaces
- Generated Static Member Functions
- Object Reference Types
- Attributes
- Any Type
- Value Type

### In addition, the following topics are discussed:

- Fixed-length Versus Variable-length User-defined Types
- Using var Classes
- Using out Classes
- Argument Passing Considerations
- Data Types
- Strings
- wchars
- wstrings
- Constants
- Enums
- Structs
- Unions
- Sequences
- Arrays
- Exceptions
- Mapping of Pseudo-objects to C++
- Usage
- Mapping Rules
- Relation to the C PIDL Mapping
- Typedefs
- Implementing Interfaces
- Implementing Operations
- PortableServer Functions
- Modules
- Interfaces
- Generated Static Member Functions
- Object Reference Types
- Attributes



- Any Type
- Value Type

# 13.1.1 Data Types

Each OMG IDL data type is mapped to a C++ data type or class.

- Basic Data Types
- Complex Data Types

# 13.1.1.1 Basic Data Types

The basic data types in OMG IDL statements are mapped to C++ typedefs in the CORBA module, as shown in the following table.

Table 13-1 Basic OMG IDL and C++ Data Types

| OMG IDL        | C++            | C++ Out Type       |
|----------------|----------------|--------------------|
| short          | CORBA::Short   | CORBA::Short_out   |
| long           | CORBA::Long    | CORBA::Long_out    |
| unsigned short | CORBA:: UShort | CORBA::UShort_out  |
| unsigned long  | CORBA::ULong   | CORBA::ULong_out   |
| float          | CORBA::Float   | CORBA::Float_out   |
| double         | CORBA::Double  | CORBA::Double_out  |
| char           | CORBA::Char    | CORBA::Char_out    |
| boolean        | CORBA::Boolean | CORBA::Boolean_out |
| octet          | CORBA::Octet   | CORBA::Octet_out   |
| wchar          | CORBA::WChar   | CORBA::WChart_out  |



On a 64-bit machine where a long integer is 64 bits, the definition of CORBA::Long would still refer to a 32-bit integer.

# 13.1.1.2 Complex Data Types

Object, pseudo-object, and user-defined types are mapped as shown in the following table.

Table 13-2 Object, Pseudo-object, and User-defined OMG IDL and C++ Types

| OMG IDL | C++               |
|---------|-------------------|
| Object  | CORBA::Object_ptr |
| struct  | C++ struct        |
| union   | C++ class         |
| enum    | C++ enum          |
| string  | char *            |
| wstring | CORBA::WChar *    |



Table 13-2 (Cont.) Object, Pseudo-object, and User-defined OMG IDL and C++ Types

| OMG IDL  | C++       |
|----------|-----------|
| sequence | C++ class |
| array    | C++ array |

The mapping for strings and UDTs is described in more detail in the following sections.

# 13.1.2 Strings

A string in OMG IDL is mapped to char \* in C++. Both bounded and unbounded strings are mapped to char \*. CORBA strings in C++ are NULL-terminated and can be used wherever a char \* type is used.

If a string is contained within another user-defined type, such as a struct, it is mapped to a CORBA::String\_var type. This ensures that each member in the struct manages its own memory.

Strings must be allocated and deallocated using the following member functions in the CORBA class:

- string alloc
- string\_dup
- string free



The string\_alloc function allocates len+1 characters so that the resulting string has enough space to hold a trailing NULL character.

# 13.1.3 wchars

OMG IDL defines a wchar data type that encodes wide characters from any character set. As with character data, an implementation is free to use any code set internally for encoding wide characters, though, again, conversion to another form may be required for transmission. The size of wchar is implementation-dependent.

The syntax for defining a wchar is:

```
<wide char type> ::= "wchar"
```

A code example for wchar is:

```
wchar t wmixed[256];
```



#### Note:

The wchar and wstring data types enable users to interact with computers in their native written language. Some languages, such as Japanese and Chinese, have thousands of unique characters. These character sets do not fit within a byte. A number of schemes have been used to support multi-byte character sets, but they have proved to be unwieldy to use. Wide characters and wide strings make it easier to interact with this kind of complexity.

# 13.1.4 wstrings

The wstring data type represents a sequence of wchar, except the wide character NULL. The type wstring is similar to that of type string, except that its element type is wchar instead of char. The actual length of a wstring is set at run time and, if the bounded form is used, must be less than or equal to the bound.

The syntax for defining a wstring is:

```
<wide_string_type> ::= "wstring" "<" <positive_int_const> ">"
| "wstring
```

A code example for wstring is:

```
CORBA::WString var v upper = CORBA::wstring dup(wmixed);
```

wstring types are built in types just like unsigned long, char, string, double, etc. They can be used directly as parameters, typedef'd, used to construct structs, sequences, unions, arrays, and so forth.

#### Note:

The wchar and wstring data types enable users to interact with computers in their native written language. Some languages, such as Japanese and Chinese, have thousands of unique characters. These character sets do not fit within a byte. A number of schemes have been used to support multi-byte character sets, but they have proved to be unwieldy to use. Wide characters and wide strings make it easier to interact with this kind of complexity.

# 13.1.5 Constants

A constant in OMG IDL is mapped to a C++ const definition. For example, consider the following OMG IDL definition:

```
// OMG IDL
const string CompanyName = "BEA Systems Incorporated";
module INVENT
{
const string Name = "Inventory Modules";
interface Order
{
```



```
const long MAX_ORDER_NUM = 10000;
};
};
```

This definition maps to C++ as follows:

```
// C++
const char *const
CompanyName = "BEA Systems Incorporated";
. . .
class INVENT
{
    static const char *const Name;
. . .
class Order : public virtual CORBA::Object
{
    static const CORBA::Long MAX_ORDER_NUM;
. . .
};
};
```

Top-level constants are initialized in the generated.h include file, but module and interface constants are initialized in the generated client stub modules.

The following is an example of a valid reference to the MAX\_ORDER\_NUM constant, as defined in the previous example:

```
CORBA::Long accnt_id = INVENT::Order::MAX_ORDER_NUM;
```

### 13.1.6 Enums

An enum in OMG IDL is mapped to an enum in C++. For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
     {
     enum Reply {ACCEPT, REFUSE};
     }
```

This definition maps to C++ as follows:



The following is an example of a valid reference to the enum defined in the previous example. You refer to enum as follows:

```
INVENT::Reply accept_reply;
accept reply = INVENT::ACCEPT;
```

### 13.1.7 Structs

A struct in OMG IDL is mapped to a C++ struct.

The generated code for a struct depends upon whether it is fixed-length or variable-length. For more information about fixed-length versus variable-length types, see the following section Fixed-length Versus Variable-length User-defined Types.

- · Fixed-length Versus Variable-length Structs
- Member Mapping
- Var
- Out

# 13.1.7.1 Fixed-length Versus Variable-length Structs

A variable-length struct contains an additional assignment operator member function to handle assignments between two variable-length structs.

For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
   // Fixed-length
   struct Date
       long year;
       long month;
       long day;
// Variable-length
struct Address
      {
      string aptNum;
      string streetName;
      string city;
      string state;
      string zipCode;
      };
};
```

This definition maps to C++ as follows:

```
// C++
class INVENT
{
```



```
struct Date
    {
        CORBA::Long year;
        CORBA::Long month;
        CORBA::Long day;
        };
        struct Address
        {
            CORBA::String_var aptNum;
            CORBA::String_var streetName;
            CORBA::String_var city;
            CORBA::String_var zipcode;
            Address &operator=(const Address &_obj);
            };
};
```

# 13.1.7.2 Member Mapping

Members of a struct are mapped to the appropriate C++ data type. For basic data types (long, short, and so on), see the following table. For object references, pseudo-object references, and strings, the member is mapped to the appropriate var class:

```
CORBA::String_varCORBA::Object var
```

All other data types are mapped as shown in the following table.

No constructor for a generated struct exists, so none of the members are initialized. Fixed-length structs can be initialized using aggregate initialization. For example:

```
INVENT::Date a_date = { 1995, 10, 12 };
```

Variable-length members map to self-managing types; these types have constructors that initialize the member.

#### 13.1.7.3 Var

A var class is generated for structs. For more information, see the section Using var Classes.

#### 13.1.7.4 Out

An out class is generated for structs. For more information, see the section Using out Classes.

### 13.1.8 Unions

A union in OMG IDL is mapped to a C++ class. The C++ class contains the following:

- Constructors
- Destructors
- Assignment operators
- · Modifiers for the union value
- Accessors for the union value



Modifiers and accessors for the union discriminator

For example, consider the following OMG IDL definition:

```
// OMG IDL
union OrderItem switch (long)
    {
    case 1: itemStruct itemInfo;
    case 2: orderStruct orderInfo;
    default: ID idInfo;
    };
```

This definition maps to C++ as follows:

```
// C++
class OrderItem
    {
    public:
      OrderItem();
      OrderItem (const OrderItem &);
       ~OrderItem();
        OrderItem & operator = (const OrderItem &);
      void d (CORBA::Long);
      CORBA::Long d () const;
      void itemInfo (const itemStruct &);
      const itemStruct & itemInfo () const;
      itemStruct & itemInfo ();
      void orderInfo (const orderStruct &);
      const orderStruct & orderInfo () const;
      orderStruct & orderInfo ();
      void idInfo (ID);
      ID idInfo () const;
};
```

The default union constructor does not set a default discriminator value for the union; therefore, you cannot call any union accessor member function until you have set the value of the union. The discriminator is an attribute that is mapped through the \_d member function.

- Union Member Accessor and Modifier Member Function Mapping
- Var
- Out
- Member Functions

## 13.1.8.1 Union Member Accessor and Modifier Member Function Mapping

For each member in the union, accessor and modifier member functions are generated.

In the following code, taken from the previous example, two member functions are generated for the ID member function:

```
void idInfo (ID);
ID idInfo () const;
```

In this example, the first function (the modifier) sets the discriminator to the default value and sets the value of the union to the specified ID value. The second function, the accessor, returns the value of the union.

Depending upon the data type of the union member, additional modifier functions are generated. The member functions generated for each data type are as follows:

 Basic data types—short, long, unsigned short, unsigned long, float, double, char, boolean, and octet

The following example generates two member functions for a basic data type with member name basictype:

```
void basictype (TYPE);  // modifier
TYPE basictype () const;  // accessor
```

For the mapping from an OMG IDL data type to the C++ data type  ${\tt TYPE}$ , see the following table.

Object and pseudo-object

For object and Typecode types with member name objtype, member functions are generated as follows:

```
void objtype (TYPE); // modifier
TYPE objtype () const; // accessor
```

For the mapping from an OMG IDL data type to the C++ data type TYPE, see the following table.

The modifier member function does not assume ownership of the specified object reference argument. Instead, the modifier duplicates the object reference or pseudo-object reference. You are responsible for releasing the reference when it is no longer required.

Enum

For an enum TYPE with member name enumtype, member functions are generated as follows:

String

For strings, one accessor and three modifier functions are generated, as follows:



The first modifier assumes ownership of the <code>char \*</code> parameter passed to it and the union is responsible for calling the <code>CORBA::string\_free</code> member function on this string when the union value changes or when the union is destroyed.

The second and third modifiers make a copy of the specified string passed in the parameter or contained in the string var.

The accessor function returns a pointer to internal memory of the union; do not attempt to free this memory, and do not access this memory after the union value has been changed or the union has been destroyed.

Struct, union, sequence, and any

For these data types, modifier and accessor functions are generated with references to the type, as follows:

The modifier function does not assume ownership of the input type parameter; instead, the function makes a copy of the data type.

Array

For an array, the modifier member function accepts an array pointer while the accessor returns a pointer to an array slice, as follows:

The modifier function does not assume ownership of the input type parameter; instead, the function makes a copy of the array.

#### 13.1.8.2 Var

A var class is generated for a union. For more information, see the section Using var Classes.

#### 13.1.8.3 Out

An out class is generated for a union. For more information, see the section Using out Classes.

### 13.1.8.4 Member Functions

In addition to the accessor and modifiers, the following member functions are generated for an OMG IDL union of type TYPE with switch (long) discriminator:

```
TYPE ();
```

This is the default constructor for a union. No default discriminator is set by this function, so you cannot access the union until you set the value of the union.

```
TYPE( const TYPE&From);
```

This copy constructor deep copies the specified union. Any data in the union parameter is copied. The From argument specifies the union to be copied.



#### ~TYPE();

This destructor frees the data associated with the union.

#### TYPE &operator=(const TYPE &From);

This assignment operator copies the specified union. Any existing value in the current union is freed. The From argument specifies the union to be copied.

```
void _d (CORBA::Long Discrim);
```

This modifier function sets the value of the union discriminant. The Discrim argument specifies the new discriminant. The data type of the argument is determined by the OMG IDL data type specified in the switch statement of the union. For each OMG IDL data type, see the following table for the C++ data type.

Only use this function to set the discriminant to a value within the same union member. You cannot use this function to implicitly switch between different union members.

These restrictions are illustrated by the following code:

```
union U switch(long) {
   case 1:
   case 2:
   short s;
   case 3:
   int it;
};

short st;
U u;
u.s(1296); // member "s" selected
   st = u.s(); // st == 1296
   u._d(2); // OK: member "s" still selected
   st = u.s(); // st == 1296
   u._d(3); // BAD_PARAM: selecting a different member
```

When the \_d() modifier is invoked on a new instance of a union, Tuxedo C++ relaxes the "implicit switching" restriction. In this case, no exception is thrown, and the union is not affected.

```
U u2;
u2._d(1);    // no exception, union is unchanged
st = u2.s(); // error! accessing an uninitialized union
u2.it(1296); // OK: member "it" now selected
```

#### CORBA::Long \_d () const;

This function returns the current discriminant value. The data type of the return value is determined by the OMG IDL data type specified in the switch statement of the union. For each OMG IDL data type, see the following table for the C++ data type.

## 13.1.9 Sequences

A sequence in OMG IDL is mapped to a C++ class. The C++ class contains the following:

- Constructors
   Each sequence has the following:
  - A default constructor



- A constructor that initializes each element
- A copy constructor
- Destructors
- Modifiers for current length (and for maximum, if the sequence is unbounded)
- Accessors for current length
- Operator[] functions to access or modify sequence elements
- Allocation and deallocation member functions

You *must* set the length before accessing any elements.

For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
{
    ...
    typedef sequence<LogItem> LogList;
}
```

This definition maps to C++ as follows:

```
// C++
class LogList
   public:
       // Default constructor
       LogList();
       // Maximum constructor
      LogList(CORBA::ULong max);
      // TYPE * data constructor
     LogList
      (
         CORBA::ULong _max,
         CORBA::ULong _length,
         LogItem * value,
         CORBA::Boolean relse = CORBA FALSE
     );
     // Copy constructor
     LogList(const LogList&);
     // Destructor
     ~LogList();
     LogList &operator=(const LogList&);
     CORBA::ULong maximum() const;
     void length(CORBA::ULong);
     CORBA::ULong length() const;
     LogItem &operator[](CORBA::ULong index);
     const LogItem &operator[](CORBA::ULong index) const;
     static LogItem *allocbuf(CORBA::ULong nelems);
     static void freebuf(LogItem *);
     };
};
```



- Sequence Element Mapping
- Vars
- Out
- Member Functions

### 13.1.9.1 Sequence Element Mapping

The <code>operator[]</code> functions are used to access or modify the sequence element. These operators return a reference to the sequence element. The OMG IDL sequence base type is mapped to the appropriate C++ data type.

For basic data types, see the following table. For object references, TypeCode references, and strings, the base type is mapped to a generated <code>\_ForSeq\_var</code> class. The <code>\_ForSeq\_var</code> class provides the capability to update a string or an object that is stored within the sequence. This generated class has the same member functions and signatures as the corresponding var class. However, this <code>\_ForSeq\_var</code> class honors the setting of the release parameter in the sequence constructor. The distinction is that the <code>\_ForSeq\_var</code> class lets users specify the <code>Release</code> flag, thereby allowing users to control the freeing of memory.

All other data types are mapped as shown in the following table.

#### 13.1.9.2 Vars

A var class is generated for a sequence. For more information, see the section Using var Classes.

#### 13.1.9.3 Out

An out class is generated for a sequence. For more information, see the section Using out Classes.

#### 13.1.9.4 Member Functions

For a given OMG IDL sequence SEQ with base type TYPE, the member functions for the generated sequence class are described as follows:

#### SEQ ();

This is the default constructor for a sequence. The length is set to 0 (zero). If the sequence is unbounded, the maximum is also set to 0 (zero). If the sequence is bounded, the maximum is specified by the OMG IDL type and cannot be changed.

#### SEQ (CORBA:: ULong Max);

This constructor is present only if the sequence is unbounded. This function sets the length of the sequence to 0 (zero) and sets the maximum of the buffer to the specified value. The Max argument specifies the maximum length of the sequence.

# SEQ (CORBA::ULong Max, CORBA::ULong Length, TYPE \* Value, CORBA::Boolean Release);

This constructor sets the maximum, length, and elements of the sequence. The Release flag determines whether elements are released when the sequence is destroyed. Explanations of the arguments are as follows:



#### Max

The maximum value of the sequence. This argument is not present in bounded sequences.

#### Length

The current length of the sequence. For bounded sequences, this value must be less than the maximum specified in the OMG IDL type.

#### Value

A pointer to the buffer containing the elements of the sequence.

#### Release

Determines whether elements are released. If this flag has a value of CORBA\_TRUE, the sequence assumes ownership of the buffer pointed to by the Value argument. If the Release flag is CORBA\_TRUE, this buffer must be allocated using the allocated member function, because it will be freed using the freebuf member function when the sequence is destroyed.

#### SEQ(const S&From);

This copy constructor deep copies the sequence from the specified argument. The From argument specifies the sequence to be copied.

#### ~SEQ();

This destructor frees the sequence and, depending upon the Release flag, may free the sequence elements.

#### SEQ&operator=(const SEQ&From);

This assignment operator deep copies the sequence from the specified sequence argument. Any existing elements in the current sequence are released if the Release flag in the current sequence is set to CORBA TRUE. The From argument specifies the sequence to be copied.

#### CORBA::ULong maximum() const;

This function returns the maximum of the sequence. For a bounded sequence, this is the value set in the OMG IDL type. For an unbounded sequence, this is the current maximum of the sequence.

#### void length(CORBA::ULong Length);

This function sets the current length of the sequence. The Length argument specifies the new length of the sequence. If the sequence is unbounded and the new length is greater than the current maximum, the buffer is reallocated and the elements are copied to the new buffer. If the new length is greater than the maximum, the maximum is set to the new length.

For a bounded sequence, the length cannot be set to a value greater than the maximum.

#### CORBA::ULong length() const;

This function returns the current length of the sequence.

# TYPE &operator[](CORBA::ULong Index); const TYPE &operator[](CORBA::ULong Index) const;

These accessor functions return a reference to the sequence element at the specified index. The Index argument specifies the index of the element to return. This index cannot be greater than the current sequence length. The length must have been set either using the  $\mathit{TYPE}$  \* constructor or the length (CORBA::ULong) modifier. If  $\mathit{TYPE}$  is an object reference, TypeCode reference, or string, the return type will be a ForSeq var class.



```
static TYPE * allocbuf(CORBA::ULong NumElems);
```

This static function allocates a buffer to be used with the TYPE \* constructor. The NumElems argument specifies the number of elements in the buffer to allocate. If the buffer cannot be allocated, NULL is returned.

If this buffer is not passed to the TYPE \* constructor with release set to CORBA\_TRUE, it must be freed using the freebuf member function.

```
static void freebuf(TYPE * Value);
```

This static function frees a TYPE \* sequence buffer allocated by the allocbuf function. The Value argument specifies the TYPE \* buffer allocated by the allocbuf function. A 0 (zero) pointer is ignored.

# 13.1.10 Arrays

An array in OMG IDL is mapped to a C++ array definition. For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
{
    . . .
    typedef LogItem LogArray[10];
};
```

This definition maps to C++ as follows:

- Array Slice
- · Array Element Mapping
- Vars
- Out
- Allocation Member Functions

## 13.1.10.1 Array Slice

A slice of an array is an array with all the dimensions of the original array except the first dimension. The member functions for the array-generated classes use a pointer to a slice to return pointers to an array. A typedef for each slice is generated.

#### For example, consider the following OMG IDL definition:

```
// OMG IDL
typedef LogItem LogMultiArray[5][10];
```

#### This definition maps to C++ as follows:

```
// C++
typedef LogItem LogMultiArray[5][10];
typedef LogItem LogMultiArray_slice[10];
```

If you have a one-dimensional array, an array slice is just a type. For example, if you had a one-dimensional array of long, an array slice would result in a CORBA::Long data type.

### 13.1.10.2 Array Element Mapping

The type of the OMG IDL array is mapped to the C++ array element type in the same manner as structs. For more information, see the section Member Mapping.

#### 13.1.10.3 Vars

A var class is generated for an array. For more information, see the section Using var Classes.

#### 13.1.10.4 Out

An out class is generated for an array. For more information, see the section Using out Classes.

### 13.1.10.5 Allocation Member Functions

For each array, there are two static functions for array allocation and deallocation. For a given OMG IDL type TYPE, the allocation and deallocation routines are as follows:

```
static TYPE slice * TYPE alloc(void);
```

This function allocates a TYPE array, returning a pointer to the allocated TYPE array. If the array cannot be dynamically allocated, 0 (zero) is returned.

```
static void TYPE free(TYPE slice * Value);
```

This function frees a dynamically allocated TYPE array. The Value argument is a pointer to the dynamically allocated TYPE array to be freed.

## 13.1.11 Exceptions

An exception in OMG IDL is mapped to a C++ class. The C++ class contains the following:

- Constructors
- Destructors
- A static narrow function, to determine the type of exception

The generated class is similar to a variable-length structure, but with an additional constructor to simplify initialization, and with the static <code>\_narrow</code> member function to determine the type of UserException.



For example, consider the following OMG IDL definition:

This definition maps to C++ as follows:

Attributes (data members) of the Exception class are public, so you may access them directly.

- Member Mapping
- Var
- Out
- Member Functions

## 13.1.11.1 Member Mapping

Members of an exception are mapped in the same manner as structs. For more information, see Member Mapping.

All exception members are public data in the C++ class, and are accessed directly.

### 13.1.11.2 Var

A var class is generated for an exception. For more information, see the section Using var Classes.



#### 13.1.11.3 Out

An out class is generated for an exception. For more information, see the section Using out Classes.

#### 13.1.11.4 Member Functions

For a given OMG IDL exception TYPE, the generated member functions are as follows:

#### static TYPE \* narrow(CORBA::Exception ptr Except);

This function returns a pointer to a TYPE exception class if the exception can be narrowed to a TYPE exception. If the exception cannot be narrowed, 0 (zero) is returned. The TYPE pointer is not a pointer to a new class. Instead, it is a typed pointer to the original exception pointer and is valid only as long as the Except parameter is valid.

#### TYPE();

This is the default constructor for the exception. No initialization of members is performed for fixed-length members. Variable-length members map to self-managing types; these types have constructors that initialize the member.

#### TYPE(member-parameters);

This constructor has an argument for each of the members in the exception. The constructor copies each argument and does not assume ownership of the memory for any argument. Building on the previous example, the signature of the constructor is:

#### NonExist (ID BadId);

There is one argument for each member of the exception. The type and parameter-passing mechanism are identical to the Any insertion operator. For information about the Any insertion operator, see the section Insertion into Any.

#### TYPE(const TYPE &From);

This copy constructor copies the data from the specified TYPE exception argument. The From argument specifies the exception to be copied.

#### ~TYPE ();

This destructor frees the data associated with the exception.

#### TYPE &operator=(const TYPE &From);

This assignment operator copies the data from the specified TYPE exception argument. The From argument specifies the exception to be copied.

#### void raise ();

This function causes the exception instance to throw itself. A catch clause

can catch it by a more derived type.

## 13.1.12 Mapping of Pseudo-objects to C++

CORBA pseudo-objects may be implemented either as normal CORBA objects or as *serverless objects*. In the CORBA specification, the fundamental differences between these strategies are:

- Serverless object types do not inherit from CORBA:: Object.
- Individual serverless objects are not registered with any ORB.



 Serverless objects do not necessarily follow the same memory management rules as for regular IDL types.

References to serverless objects are not necessarily valid across computational contexts; for example, address spaces. Instead, references to serverless objects that are passed as parameters may result in the construction of independent, functionally identical copies of objects used by receivers of these references. To support this, the otherwise hidden representational properties (such as data layout) of serverless objects are made known to the ORB. Specifications for achieving this are not contained in this chapter; making serverless objects known to the ORB is an implementation detail.

This chapter provides a standard mapping algorithm for all pseudo-object types. This avoids the need for piecemeal mappings for each of the nine CORBA pseudo-object types, and accommodates any pseudo-object types that may be proposed in future revisions of *CORBA*. It also avoids representation dependence in the C mapping, while still allowing implementations that rely on C-compatible representations.

## 13.1.13 Usage

Rather than C-PIDL, this mapping uses an augmented form of full OMG IDL to describe serverless object types. Interfaces for pseudo-object types follow the same rules as normal OMG IDL interfaces, with the following exceptions:

- They are prefaced by the keyword pseudo.
- Their declarations may refer to other <sup>1</sup> serverless object types that are not otherwise necessarily allowed in OMG IDL.

The pseudo prefix means that the interface may be implemented in either a normal or serverless fashion. That is, apply either the rules described in the following sections, or the normal mapping rules described in this chapter.

## 13.1.14 Mapping Rules

Serverless objects are mapped in the same way as normal interfaces, except for the differences outlined in this section.

Classes representing serverless object types are *not* subclasses of CORBA::Object, and are not necessarily subclasses of any other C++ class. Thus, they do not necessarily support, for example, the Object::create request operation.

For each class representing a serverless object type  ${\tt T}$ , overloaded versions of the following functions are provided in the CORBA namespace:

```
// C++
void release(T_ptr);
Boolean is_nil(T_ptr p);
```

The mapped C++ classes are not guaranteed to be usefully subclassable by users, although subclasses can be provided by implementations. Implementations are allowed to make assumptions about internal representations and transport formats that may not apply to subclasses.

The member functions of classes representing serverless object types do not necessarily obey the normal memory management rules. This is because some serverless objects, such as CORBA::NVList, are essentially just containers for several levels of other serverless objects.



Requiring callers to explicitly free the values returned from accessor functions for the contained serverless objects would be counter to their intended usage.

In particular, exception used as a data type and a function name. All other elements of the mapping are the same.

- The types of references to serverless objects, T\_ptr, may or may not simply be a typedef
  of T\*.
- Each mapped class supports the following static member functions:

```
// C++
static T_ptr _duplicate(T_ptr p);
static T ptr _nil();
```

- Legal implementations of \_duplicate include simply returning the argument or constructing references to a new instance. Individual implementations may provide stronger guarantees about behavior.
- The corresponding C++ classes may or may not be directly instantiable or have other instantiation constraints. For portability, users must invoke the appropriate constructive operations.
- As with normal interfaces, assignment operators are not supported.
- Although they can transparently employ "copy-style" rather than "reference-style"
  mechanics, parameter passing signatures and rules as well as memory management rules
  are identical to those for normal objects, unless otherwise noted.

## 13.1.15 Relation to the C PIDL Mapping

All serverless object interfaces and declarations that rely on them have direct analogs in the C mapping. The mapped C++ classes can, but need not, be implemented using representations compatible to those chosen for the C mapping. Differences between the pseudo-object specifications for C-PIDL and C++ PIDL are as follows:

- C++ PIDL calls for removal of representation dependencies through the use of interfaces rather than structs and typedefs.
- C++ PIDL calls for placement of operations on pseudo-objects in their interfaces, including a few cases of redesignated functionality as noted.
- In C++ PIDL, release performs the role of the associated free and delete operations in the C mapping, unless otherwise noted.

Brief descriptions and listings of each pseudo-interface and its C++ mapping are provided in the following sections. Further details, including definitions of types referenced but not defined below, may be found in the relevant sections of this document.

## 13.1.16 Typedefs

A typedef in OMG IDL is mapped to a typedef in C++. Depending upon the OMG IDL data type, additional typedefs and member functions may be defined. The generated code for each data type is as follows:

 Basic data types (short, long, unsigned short, unsigned long, float, double, char, boolean, and octet)



Basic data types map to a simple typedef. For example:

```
// OMG IDL
typedef long ID;
// C++
typedef CORBA::Long ID;
```

string

A string typedef is mapped to a simple typedef. For example:

```
// OMG IDL
typedef string IDStr;
// C++
typedef char * IDStr;
```

object, interfaces, TypeCode

Object, interfaces, and TypeCode types are mapped to four typedefs. For example:

```
// OMG IDL
typedef Item Intf;

// C++
typedef Item Intf;
typedef Item_ptr Intf_ptr;
typedef Item_var Intf_var;
typedef Item_ptr &Intf_ out;
```

enum, struct, union, sequence
 UDTs are mapped to three typedefs. For example:

```
// OMG IDL
typedef LogList ListRetType;

// C++
typedef LogList ListRetType;
typedef LogList_var ListRetType_var;
typedef LogList out &ListRetType out;
```

array

Arrays are mapped to four typedefs and the static member functions to allocate and free memory. For example:

```
// OMG IDL
typedef LogArray ArrayRetType;

// C++
typedef LogArray ArrayRetType;
typedef LogArray_var ArrayRetType_var;
typedef LogArray_forany ArrayRetType_forany;
typedef LogArray_slice ArrayRetType_slice;
ArrayRetType_slice * ArrayRetType_alloc();
void ArrayRetType free(ArrayRetType slice *);
```



## 13.1.17 Implementing Interfaces

An operation in OMG IDL is mapped to a C++ member function.

The name of the member function is the name of the operation. The operation is defined as a member function in both the interface class and the stub class. The interface class is virtual; the stub class inherits from the virtual class and contains the member function code from the client application stub. When an operation is invoked on the object reference, the code contained in the corresponding stub member function executes.

For example, consider the following OMG IDL definition:

This definition maps to C++ as follows:

The generated client application stub then contains the following generated code for the stub class:

```
// ROUTINE NAME: INVENT::Stub_Order::modifyOrder
//
// FUNCTIONAL DESCRIPTION:
//
// Client application stub routine for operation
// modifyOrder.
// (Interface : Order)
INVENT::ItemList * INVENT::Stub_Order::modifyOrder (
```



```
const INVENT::ItemList & ModifyList)
{
    . . .
}
```

Argument Mapping

## 13.1.17.1 Argument Mapping

Each of the arguments in an operation is mapped to the corresponding C++ type as described in the following table Table 13-1 and table Table 13-2.

The parameter passing modes for arguments in an operation are described in the following table Table 13-8 and table Table 13-9.

## 13.1.18 Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client side, the server-side mapping requires that the function header include the appropriate exception (throw) specification. This requirement allows the compiler to detect when an invalid exception is raised, which is necessary in the case of a local C++-to-C+ + library call (otherwise, the call would have to go through a wrapper that checks for a valid exception). For example:

```
// IDL
interface A
{
  exception B {};
  void f() raises(B);
};
// C++
  class MyA : public virtual POA_A
{
    public:
  void f() throw(A::B, CORBA::SystemException);
  ...
};
```

Since all operations and attributes may throw CORBA system exceptions, CORBA::SystemException must appear in all exception specifications, even when an operation

CORBA::SystemException must appear in all exception specifications, even when an operation has no raises clause.

Within a member function, the "this" pointer refers to the implementation object's data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. For example:

```
// IDL
interface A
{
void f();
void g();
};
// C++
class MyA : public virtual POA_A
{
```



```
public:
void f() throw(SystemException);
void g() throw(SystemException);
private:
long x_;
};
void
MyA::f() throw(SystemException)
{
this->x_ = 3;
this->g();
}
```

However, when a servant member function is invoked in this manner, it is being called simply as a C++ member function, not as the implementation of an operation on a CORBA object. In such a context, any information available via the POA\_Current object refers to the CORBA request invocation that performed the C++ member function invocation, not to the member function invocation itself.

Skeleton Derivation from Object

## 13.1.18.1 Skeleton Derivation from Object

In several existing ORB implementations, each skeleton class derives from the corresponding interface class. For example, for interface Mod::A, the skeleton class  $POA\_Mod::A$  is derived from class Mod::A. These systems, therefore, allow an object reference for a servant to be implicitly obtained via normal C++ derived-to-base conversion rules:

Such code can be supported by a conforming ORB implementation, but it is not required, and is thus not portable. The equivalent portable code invokes \_this() on the implementation object to implicitly register it if it has not yet been registered, and to get its object reference:

### 13.1.19 PortableServer Functions

Objects registered with POAs use sequences of octet, specifically the PortableServer::POA::ObjectId type, as object identifiers. However, because C++ programmers often want to use strings as object identifiers, the C++ mapping provides several conversion functions that convert strings to ObjectId and vice versa:

```
/ C++
namespace PortableServer
{
char* ObjectId_to_string(const ObjectId&);
```



```
ObjectId* string_to_ObjectId(const char*);
}
```

These functions follow the normal C++ mapping rules for parameter passing and memory management.

If conversion of an ObjectId to a string would result in illegal characters in the string (such as a NULL), the first two functions throw the CORBA::BAD PARAM exception.

### 13.1.20 Modules

A module in OMG IDL is mapped to a C++ class. Objects contained in the module are defined within this C++ class. Because interfaces and types are also mapped to classes, nested C++ classes result.

For example, consider the following OMG IDL definition:

```
// OMG IDL
module INVENT
{
interface Order
{
          . . .
};
};
```

This definition maps to C++ as follows:

Multiple nested modules yield multiple nested classes. Anything inside the module will be in the module class. Anything inside the interface will be in the interface class.

OMG IDL allows modules, interfaces, and types to have the same name. However, when generating files for the C++ language, having the same name is not allowed. This restriction is necessary because the OMG IDL names are generated into nested C++ classes with the same name; this is not supported by C++ compilers.





The Oracle Tuxedo OMG IDL compiler outputs an informational message if you generate C++ code from OMG IDL with an interface or type with the same name as the current module name. If you ignore this informational message and do not use unique names to differentiate the interface or type from the module name, the compiler will signal errors when compiling the generated files.

### 13.1.21 Interfaces

An interface in OMG IDL is mapped to a C++ class. This class contains the definitions of the operations, attributes, constants, and user-defined types (UDTs) contained in the OMG IDL interface.

For an interface *INTF*, the generated interface code contains the following items:

- Object reference type (INTF ptr)
- Object reference variable type (INTF var)
- duplicate static member function
- narrow static member function
- nil static member function
- UDTs
- Member functions for attributes and operations

For example, consider the following OMG IDL definition:

This definition maps to C++ as follows:



```
};
};
```

The object reference types and static member functions are described in the following sections, as are UDTs, operations, and attributes.

## 13.1.22 Generated Static Member Functions

This section describes in detail the generated static member functions: \_duplicate, \_narrow, and \_nil for an interface INTF.

```
static INTF ptr duplicate (INTF ptr Obj)
```

This static member function duplicates an existing *INTF* object reference and returns a new *INTF* object reference. The new *INTF* object reference must be released by calling the CORBA::release member function. If an error occurs, a reference to the nil *INTF* object is returned. The argument Obj specifies the object reference to be duplicated.

```
static INTF ptr narrow (CORBA::Object ptr Obj)
```

This static member function returns a new *INTF* object reference given an existing CORBA::Object\_ptr object reference. The Object\_ptr object reference may have been created by a call to the CORBA::ORB::string\_to\_object member function or may have been returned as a parameter from an operation.

The <code>INTF</code> \_ptr object reference must correspond to an <code>INTF</code> object or to an object that inherits from the <code>INTF</code> object. The new <code>INTF</code> object reference must be released by calling the <code>CORBA::release member</code> function. The argument <code>Obj</code> specifies the object reference to be narrowed to an <code>INTF</code> object reference. The <code>Obj</code> parameter is not modified by this member function and must be released by the user when it is no longer required. If <code>Obj</code> cannot be narrowed to an <code>INTF</code> object reference, the <code>INTF</code> nil object reference is returned.

```
static INTF ptr nil ( )
```

This static member function returns the new nil object reference for the *INTF* interface. The new reference does *not* have to be released by calling the CORBA::release member function.

## 13.1.23 Object Reference Types

An interface class (INTF) is a virtual class; the CORBA standard does not allow you to:

- Create or hold an instance of the interface class
- Use a pointer or a reference to the interface class

Instead, you use one of the object reference types, INTF ptr or INTF var class.

You can obtain an object reference by using the \_narrow static member function. Operations are invoked on these classes using the arrow operator (->).

The <code>INTF\_var</code> class simplifies memory management by automatically releasing the object reference when the <code>INTF\_var</code> class goes out of scope or is reassigned. Variable types are generated for many of the UDTs and are described in Using var Classes.

## 13.1.24 Attributes

A read-only attribute in OMG IDL is mapped to a C++ function that returns the attribute value. A read-write attribute maps to two overloaded C++ functions, one to return the attribute value

and one to set the attribute value. The name of the overloaded member function is the name of the attribute.

Attributes are generated in the same way that operations are generated. They are defined in both the virtual and the stub classes. For example, consider the following OMG IDL definition:

This definition maps to C++ as follows:

The generated client application stub then contains the following generated code for the stub class:

```
// ROUTINE NAME: INVENT::Stub_Item::itemInfo
//
// FUNCTIONAL DESCRIPTION:
//
// Client application stub routine for attribute
// INVENT::Stub_Item::itemInfo. (Interface : Item)
INVENT::itemStruct * INVENT::Stub_Item::itemInfo ( )
{
. . . .
}
// ROUTINE NAME: INVENT::Stub_Item::itemInfo
//
// FUNCTIONAL DESCRIPTION:
```



```
//
// Client application stub routine for attribute
// INVENT::Stub_Item::itemInfo. (Interface : Item)
void INVENT::Stub_Item::itemInfo (
   const INVENT::itemStruct & itemInfo)
{
}
```

Argument Mapping

### 13.1.24.1 Argument Mapping

An attribute is equivalent to two operations, one to return the attribute and one to set the attribute. For example, the itemInfo attribute listed above is equivalent to:

```
void itemInfo (in itemStruct itemInfo);
itemStruct itemInfo ();
```

The argument mapping for the attribute is the same as the mapping for an operation argument. The attribute is mapped to the corresponding C++ type as described in the following table Basic OMG IDL and C++ Data Types and table Object, Pseudo-object, and User-defined OMG IDL and C++ Types. The parameter passing modes for arguments in an operation are described in table Basic Argument and Result Passing and table T\_var Argument and Result Passing.

## 13.1.25 Any Type

An any in OMG IDL is mapped to the CORBA:: Any class. The CORBA:: Any class handles C++ types in a type-safe manner.

- Handling Typed Values
- Insertion into Any
- Extraction from Any
- Distinguishing Boolean, Octet, Char, and Bounded Strings
- Widening to Object
- Handling Untyped Values
- Any Constructors, Destructor, Assignment Operator
- The Any Class

## 13.1.25.1 Handling Typed Values

To decrease the chances of creating an any with a mismatched TypeCode and value, the C++ function overloading facility is utilized. Specifically, for each distinct type in an OMG IDL specification, overloaded functions to insert and extract values of that type are provided. Overloaded operators are used for these functions to completely avoid any name space pollution. The nature of these functions, which are described in detail below, is that the appropriate TypeCode is implied by the C++ type of the value being inserted into or extracted from the any.



Since the type-safe any interface described below is based upon C++ function overloading, it requires C++ types generated from OMG IDL specifications to be distinct. However, there are special cases in which this requirement is not met:

- The Boolean, octet, and char OMG IDL types are not required to map to distinct C++ types, which means that a separate means of distinguishing them from each other for the purpose of function overloading is necessary. The means of distinguishing these types from each other is described in Distinguishing Boolean, Octet, Char, and Bounded Strings.
- Since all strings are mapped to char\* regardless of whether they are bounded or unbounded, another means of creating or setting an any with a bounded string value is necessary.
- In C++, arrays within a function argument list decay into pointers to their first elements.
  This means that function overloading cannot be used to distinguish between arrays of
  different sizes. The means for creating or setting an any when dealing with arrays is
  described below and in Arrays.

### 13.1.25.2 Insertion into Any

To allow a value to be set in an any in a type-safe fashion, the following overloaded operator function is provided for each separate OMG IDL type T:

```
// C++
void operator<<=(Any&, T);
```

This function signature suffices for the following types, which are usually passed by value:

- Short, UShort, Long, ULong, Float, Double
- Enumerations
- Unbounded strings (char\* passed by value)
- Object references (T ptr)

For values of type T that are too large to be passed by value efficiently, two forms of the insertion function are provided:

The copying form is largely equivalent to the first form shown, as far as the caller is concerned.

These "left-shift-assign" operators are used to insert a typed value into an any, as follows:

```
// C++
Long value = 42;
Any a;
a <<= value;</pre>
```

In this case, the version of operator<<= overloaded for type Long sets both the value and the TypeCode properly for the Any variable.

Setting a value in an any using operator<<= means the following:



- For the copying version of operator<<=, the lifetime of the value in the Any is independent of the lifetime of the value passed to operator<<=. The implementation of the Any does not store its value as a reference or a pointer to the value passed to operator<<=.
- For the noncopying version of operator<<=, the inserted  $\mathbb{T}^*$  is consumed by the Any. The caller may not use the  $\mathbb{T}^*$  to access the pointed-to data after insertion because the Any assumes ownership of  $\mathbb{T}^*$ , and the Any may immediately copy the pointed-to data and destroy the original.
- With both the copying and noncopying versions ofoperator<<=, any previous value held by
  the Any is properly deallocated. For example, if the Any (TypeCode\_ptr, void\*, TRUE)
  constructor (described in Handling Untyped Values) were called to create the Any, the Any
  is responsible for deallocating the memory pointed to by the void\* before copying the new
  value.</li>

Copying insertion of a string type causes the following function to be invoked:

```
// C++
void operator<<=(Any&, const char*);</pre>
```

Since all string types are mapped to <code>char\*</code>, this insertion function assumes that the value being inserted is an unbounded string. Distinguishing Boolean, Octet, Char, and Bounded Strings describes how bounded strings may be correctly inserted into an <code>Any</code>. Noncopying insertion of both bounded and unbounded strings can be achieved using the <code>Any::from\_string</code> helper type described in Distinguishing Boolean, Octet, Char, and Bounded Strings.

Type-safe insertion of arrays uses the Array\_forany types described in Arrays. The ORB provides a version of operator<<= overloaded for each Array forany type. For example:

```
// IDL
typedef long LongArray[4][5];

// C++
typedef Long LongArray[4][5];
typedef Long LongArray_slice[5];
class LongArray_forany { ... };

void operator<<=(Any &, const LongArray_forany &);</pre>
```

The Array\_forany types are always passed to operator<<= by reference to const. The nocopy flag in the Array\_forany constructor is used to control whether the inserted value is copied (nocopy == FALSE) or consumed (nocopy == TRUE). Because the nocopy flag defaults to FALSE, copying insertion is the default.

Because of the type ambiguity between an array of  $\mathbb{T}$  and a  $\mathbb{T}^*$ , it is highly recommended that portable code explicitly use the appropriate  $\texttt{Array\_forany}$  type when inserting an array into an Any. For example:

```
// IDL
struct S {... };
typedef S SA[5];
// C++
struct S { ... };
typedef S SA[5];
typedef S SA slice;
```



Line 1 results in the invocation of the noncopying <code>operator<<=(Any&, S\*)</code> due to the decay of the <code>SA</code> array type into a pointer to its first element, rather than the invocation of the copying <code>SA\_forany</code> insertion operator. Line 2 explicitly constructs the <code>SA\_forany</code> type and thus results in the desired insertion operator being invoked.

The noncopying version of operator << for object references takes the address of the  $T_ptr$  type, as follows:

The noncopying object reference insertion consumes the object reference pointed to by  $\mathbb{T}_ptr^*$ ; therefore, after insertion the caller may not access the object referred to by  $\mathbb{T}_ptr$  because the Any may have duplicated and then immediately released the original object reference. The caller maintains ownership of the storage for the  $\mathbb{T}_ptr$  itself.

The copying version of operator<<= is also supported on the Any var type.

### 13.1.25.3 Extraction from Any

To allow type-safe retrieval of a value from an any, the ORB provides the following operators for each OMG IDL type T:

```
// C++
Boolean operator>>=(const Any&, T&);
```

This function signature suffices for primitive types that are usually passed by value. For values of type T that are too large to be passed by value efficiently, the ORB provides a different signature, as follows:

```
// C++
Boolean operator>>=(const Any&, T*&);
```

The first form of this function is used only for the following types:

- Boolean, Char, Octet, Short, UShort, Long, ULong, Float, Double
- Enumerations
- Unbounded strings (char\* passed by reference, i.e., char\*&)
- Object references (T ptr)

For all other types, the second form of the function is used.



This "right-shift-assign" operator is used to extract a typed value from an any, as follows:

In this case, the version of operator>>= for type Long determines whether the Any truly does contain a value of type Long and, if so, copies its value into the reference variable provided by the caller and returns TRUE. If the Any does not contain a value of type Long, the value of the caller's reference variable is not changed, and operator>>= returns FALSE.

For nonprimitive types, extraction is done by pointer. For example, consider the following OMG IDL struct:

```
// IDL
struct MyStruct {
    long lmem;
    short smem;
};
```

Such a struct could be extracted from an Any as follows:

If the extraction is successful, the caller's pointer points to storage managed by the Any, and operator>>= returns TRUE. The caller must not try to delete or otherwise release this storage. The caller also must not use the storage after the contents of the Any variable are replaced via assignment, insertion, or the replace function, or after the Any variable is destroyed. Care must be taken to avoid using  $T_{var}$  types with these extraction operators, since they will try to assume responsibility for deleting the storage owned by the Any.

If the extraction is not successful, the value of the caller's pointer is set equal to the NULL pointer, and operator>>= returns FALSE.

Correct extraction of array types relies on the Array forany types described in Arrays.

An example of the OMG IDL is as follows:

```
// IDL
typedef long A[20];
typedef A B[30][40][50];
// C++
typedef Long A[20];
typedef Long A_slice;
class A forany { ... };
```



The Array forany types are always passed to operator>>= by reference.

For strings and arrays, applications are responsible for checking the TypeCode of the Any to be sure that they do not overstep the bounds of the array or string object when using the extracted value.

The operator>>= is also supported on the Any var type.

## 13.1.25.4 Distinguishing Boolean, Octet, Char, and Bounded Strings

Since the Boolean, octet, and char OMG IDL types are not required to map to distinct C++ types, another means of distinguishing them from each other is necessary so that they can be used with the type-safe Any interface. Similarly, since both bounded and unbounded strings map to char\*, another means of distinguishing them must be provided. This is done by introducing several new helper types nested in the Any class interface. For example, this is accomplished as shown below:

```
// C++
class Any
 public:
   // special helper types needed for boolean, octet,
   // char, and bounded string insertion
   struct from boolean {
       from boolean(Boolean b) : val(b) {}
       Boolean val;
   };
   struct from octet {
        from octet(Octet o) : val(o) {}
        Octet val;
   };
   struct from char {
   from char(Char c) : val(c) {}
   Char val;
   };
   struct from string {
     from string(char* s, ULong b,
               Boolean nocopy = FALSE) :
            val(s), bound(b) {}
     char *val;
     ULong bound;
   };
   void operator<<=(from boolean);</pre>
   void operator<<=(from char);</pre>
   void operator<<=(from octet);</pre>
  void operator<<=(from string);</pre>
  // special helper types needed for boolean, octet,
 // char, and bounded string extraction
```

```
struct to boolean {
    to boolean (Boolean &b) : ref(b) {}
   Boolean &ref;
  };
  struct to char {
     to char(Char &c) : ref(c) {}
     Char &ref;
  };
  struct to octet {
     to_octet(Octet &o) : ref(o) {}
     Octet &ref;
  };
  struct to string {
     to string(char *&s, ULong b) : val(s), bound(b) {}
     char *&val;
     ULong bound;
  };
 Boolean operator>>=(to boolean) const;
 Boolean operator>>=(to char) const;
 Boolean operator>>=(to octet) const;
 Boolean operator>>=(to string) const;
  // other public Any details omitted
private:
  // these functions are private and not implemented
 // hiding these causes compile-time errors for
 // unsigned char
void operator<<=(unsigned char);</pre>
Boolean operator>>=(unsigned char &) const;
};
```

The ORB provides the overloaded <code>operator<<=</code> and <code>operator>>=</code> functions for these special helper types. These helper types are used as shown here:

A bound value of 0 (zero) indicates an unbounded string.

For noncopying insertion of a bounded or unbounded string into an Any, the nocopy flag on the from string constructor must be set to TRUE:

```
// C++
char* p = string_alloc(8);
```

```
// ...initialize string p...
any <<= Any::from string(p, 8, 1);  // any consumes p</pre>
```

Assuming that boolean, char, and octet all map the C++ type unsigned char, the private and unimplemented operator<<= and operator>>= functions for unsigned char cause a compile-time error if straight insertion or extraction of any of the boolean, char, or octet types is attempted:

## 13.1.25.5 Widening to Object

Sometimes it is desirable to extract an object reference from an Any as the base Object type. This can be accomplished using a helper type similar to those required for extracting boolean, char, and octet:

```
// C++
class Any
{
   public:
     ...
     struct to_object {
        to_object(Object_ptr &obj) : ref(obj) {}
        Object_ptr &ref;
    ;
      Boolean operator>>=(to_object) const;
    ...
};
```

The to\_object helper type is used to extract an object reference from an Any as the base Object type. If the Any contains a value of an object reference type as indicated by its TypeCode, the extraction function operator>>=  $(to_object)$  explicitly widens its contained object reference to Object and returns TRUE; otherwise, it returns FALSE. This is the only object reference extraction function that performs widening on the extracted object reference. As with regular object reference extraction, no duplication of the object reference is performed by the to\_object extraction operator.

### 13.1.25.6 Handling Untyped Values

Under some circumstances the type-safe interface to Any is not sufficient. An example is a situation in which data types are read from a file in binary form and are used to create values of type Any. For these cases, the Any class provides a constructor with an explicit TypeCode and generic pointer:

```
// C++
Any(TypeCode ptr tc, void *value, Boolean release = FALSE);
```

The constructor duplicates the given TypeCode pseudo-object reference. If the release parameter is TRUE, the Any object assumes ownership of the storage pointed to by the value

parameter. A caller must make no assumptions about the continued lifetime of the value parameter once it has been handed to an Any with release=TRUE, since the Any may copy the value parameter and immediately free the original pointer. If the release parameter is FALSE (the default case), the Any object assumes that the caller manages the memory pointed to by value. The value parameter can be a NULL pointer.

The Any class also defines three unsafe operations:

```
// C++
void replace(
    TypeCode_ptr,
    void *value,
    Boolean release = FALSE
);
TypeCode_ptr type() const;
const void *value() const;
```

The replace function is intended to be used with types that cannot be used with the type-safe insertion interface, and so is similar to the constructor described above. The existing TypeCode is released and value storage is deallocated, if necessary. The TypeCode function parameter is duplicated. If the release parameter is TRUE, the Any object assumes ownership for the storage pointed to by the value parameter. The Any must make no assumptions about the continued lifetime of the value parameter once it has been handed to the Any::replace function with release=TRUE, since the Any may copy the value parameter and immediately free the original pointer. If the release parameter is FALSE (the default case), the Any object assumes that the caller manages the memory occupied by the value. The value parameter of the replace function can be a NULL pointer.

#### Note:

Neither the constructor shown above nor the replace function is type-safe. In particular, no guarantees are made by the compiler at run time as to the consistency between the TypeCode and the actual type of the void\* argument. The behavior of an ORB implementation when presented with an Any that is constructed with a mismatched TypeCode and value is not defined.

The type function returns a  $TypeCode_ptr$  pseudo-object reference to the TypeCode associated with the Any. Like all object reference return values, the caller must release the reference when it is no longer needed, or assign it to a  $TypeCode_var$  variable for automatic management.

The value function returns a pointer to the data stored in the Any. If the Any has no associated value, the value function returns a NULL pointer.

### 13.1.25.7 Any Constructors, Destructor, Assignment Operator

The default constructor creates an Any with a TypeCode of type <code>tk\_null</code>, and no value. The copy constructor calls <code>\_duplicate</code> on the <code>TypeCode\_ptr</code> of its Any parameter and deep-copies the parameter's value. The assignment operator releases its own <code>TypeCode\_ptr</code> and deallocates storage for the current value if necessary, then duplicates the <code>TypeCode\_ptr</code> of its Any parameter and deep-copies the parameter's value. The destructor calls <code>release</code> on the <code>TypeCode\_ptr</code> and deallocates storage for the value, if necessary.



Other constructors are described in the section Handling Untyped Values.

## 13.1.25.8 The Any Class

The full definition of the Any class can be found in the section Any Class Member Functions.

## 13.1.26 Value Type

This section is based on information contained in Chapters 3, 5, and 6 of the *Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, February 2001, and the *CORBA C++ Language Mapping Specification*, June 1999, published by the Object Management Group (OMG). Used with permission of the OMG.

- Overview
- Architecture
- Benefits
- Valuetype Example

#### 13.1.26.1 Overview

Objects, more specifically, interface types that objects support, are defined in an IDL interface, allowing arbitrary implementations. There is great value in having a distributed object system that places almost no constraints on implementation. However, there are many occasions in which it is desirable to be able to pass an object by value, rather than by reference. This may be particularly useful when an object's primary "purpose" is to encapsulate data, or an application explicitly wishes to make a "copy" of an object.

The semantics of passing an object by value are similar to that of standard programming languages. The receiving side of a parameter passed by value receives a description of the "state" of the object. It then instantiates a new instance with that state but having a separate identity from that of the sending side. Once the parameter passing operation is complete, no relationship is assumed to exist between the two instances.

Because it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state and implementation. Thus, valuetype(s) provide semantics that bridge between CORBA structs and CORBA interfaces, as follows:

- They support description of complex state (that is, arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call).
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation (that is, they can support an interface).
- They support single inheritance (of valuetype) and can support an interface.
- They may be also be abstract.

#### 13.1.26.2 Architecture

The basic notion of valuetypes is relatively simple. A valuetype is, in some sense, half way between a "regular" IDL interface type and a struct. The use of valuetype is a signal from the application programmer that some additional properties (state) and implementation details be



specified beyond that of an interface type. Specification of this information puts some additional constraints on the implementation choices beyond that of interface types. This is reflected in both the semantics specified herein, and in the language mappings.

#### 13.1.26.3 Benefits

Prior to supporting valuetypes (objects passable by value), all CORBA objects had object references. When multiple clients invoked on a particular object, they use the same object reference. The instance(s) of the object remained on the server ORB and its state was maintained by the server ORB, not the client ORB.

Valuetypes represent a significant addition to the CORBA architecture. As with objects passed by reference, valuetypes have state and methods, but do not have object references and are always invoked locally as programming language objects. Upon request from the receiving side, valuetypes package their state in the sending context, send their state "over the wire" to the receiving side, where an instance is created and populated with the transmitted state. The sending side has no further control of the client-side instance. Thus, the receiving side can make subsequent invocations of the instance locally. This model eliminates the delays involved when communicating over the network. These delays can be significant in large networks. The addition of valuetypes enables CORBA implementations to more easily scale to meet large data-handling requirements.

Therefore, an essential property of valuetypes is that their implementations are always local. That is, the explicit use of valuetypes in a concrete programming language is always guaranteed to use a local implementation, and will not require a remote call. They have no identity (their value is their identity) and they are not "registered" with the ORB.

## 13.1.26.4 Valuetype Example

For example, consider the following IDL valuetype taken from the *CORBA C++ Language Mapping Specification*, June 1999, published by the Object Management Group (OMG):

```
// IDL
valuetype Example {
      short op1();
      long op2(in Example x);
private short val1;
public long val2;
private string val3;
private float val4;
private Example val5;
};
The C++ mapping for this valuetype is:
// C++
class Example : public virtual ValueBase {
public:
   virtual Short op1() = 0;
   virtual Long op2(Example*) = 0;
   virtual Long val2() const = 0;
   virtual void val2(Long) = 0;
    static Example* downcast(ValueBase*);
protected:
  Example();
  virtual ~Example();
  virtual Short val1() const = 0;
  virtual void val1(Short) = 0;
```



```
virtual const char* val3() const = 0;
  virtual void val3(char*) = 0;
  virtual void val3(const char*) = 0;
  virtual void val3(const String var&) = 0;
  virtual Float val4() const = 0;
   virtual void val4(Float) = 0;
  virtual Example* val5() const = 0;
  virtual void val5(Example*) = 0;
private:
  // private and unimplemented
  void operator=(const Example&);
class OBV Example : public virtual Example {
   public:
      virtual Long val2() const;
      virtual void val2(Long);
protected:
      OBV Example();
      OBV_Example(Short init_val1, Long init_val2,
          const char* init val3, Float init val4,
          Example* init val5);
      virtual ~OBV Example();
      virtual Short val1() const;
      virtual void val1(Short);
      virtual const char* val3() const;
      virtual void val3(char*);
      virtual void val3(const char*);
      virtual void val3(const String var&);
      virtual Float val4() const;
      virtual void val4(Float);
      virtual Example* val5() const;
      virtual void val5(Example*);
      // ...
};
```

# 13.2 Fixed-length Versus Variable-length User-defined Types

The memory management rules and member function signatures for a user-defined type depend upon whether the type is fixed-length or variable-length. A user-defined type is variable-length if it is one of the following:

- A bounded or unbounded string
- A bounded or unbounded sequence
- A struct or union that contains a variable-length member
- An array with a variable-length element type
- A typedef to a variable-length type

If a type is not on this list, the type is fixed-length.

# 13.3 Using var Classes

Automatic variables (vars) are provided to simplify memory management. Vars are provided through a var class that assumes ownership for the memory required for the type and frees the memory when the instance of the var object is destroyed or when a new value is assigned to the var object.

The Oracle Tuxedo provides var classes for the following types:

- String (CORBA::String var)
- Object references (CORBA::Object var)
- User-defined OMG IDL types (struct, union, sequence, array, and interface)

The var classes have common member functions, but may support additional operators depending upon the OMG IDL type. For an OMG IDL type TYPE, the TYPE\_var class contains constructors, destructors, assignment operators, and operators to access the underlying TYPE type. An example var class is as follows:

```
class TYPE var
    public:
        // constructors
        TYPE var();
        TYPE var(TYPE *);
        TYPE var(const TYPE_var &);
        // destructor
        ~TYPE_var();
        // assignment operators
        TYPE var &operator=(TYPE *);
        TYPE var &operator=(const TYPE var &);
       // accessor operators
       TYPE *operator->();
       TYPE *operator->() const;
       TYPE var ptr in() const;
       TYPE var ptr& inout();
       TYPE var ptr& out();
       TYPE_var_ptr _retn();
       operator const TYPE ptr&() const;
       operator TYPE ptr&();
       operator TYPE ptr;
       };
```

The details of the member functions are as follows:

#### TYPE\_var()

This is the default constructor for the <code>TYPE\_var</code> class. The constructor initializes to 0 (zero) the <code>TYPE \*</code> owned by the var class. You may not invoke the <code>operator-></code> on a <code>TYPE\_var</code> class unless a valid <code>TYPE \*</code> has been assigned to it.

#### TYPE var(TYPE \* Value);

This constructor assumes ownership of the specified TYPE \* parameter. When the TYPE\_var is destroyed, the TYPE is released. The Value argument is a pointer to the TYPE to be owned by this var class. This pointer must not be 0 (zero).

#### TYPE var(const TYPE var &From);

This copy constructor allocates a new TYPE and makes a deep copy of the data contained in the TYPE owned by the From parameter. When the  $TYPE\_var$  is destroyed, the copy of the TYPE is released or deleted. The From parameter specifies the var class that points to the TYPE to be copied.

#### ~TYPE var();

This destructor uses the appropriate mechanism to release the TYPE owned by the var class. For strings, this is the <code>CORBA::string\_free</code> routine. For object references, this is the <code>CORBA::release</code> routine. For other types, this may be <code>delete</code> or a generated static routine used to free allocated memory.

#### TYPE var &operator=(TYPE \* NewValue);

This assignment operator assumes ownership of the TYPE pointed to by the NewValue parameter. If the TYPE\_var currently owns a TYPE, it is released before assuming ownership of the NewValue parameter. The NewValue argument is a pointer to the TYPE to be owned by this var class. This pointer must not be 0 (zero).

#### TYPE var &operator=(const TYPE\_var &From);

This assignment operator allocates a new TYPE and makes a deep copy of the data contained in the TYPE owned by the From TYPE\_var parameter. If TYPE\_var currently owns a TYPE, it is released. When the TYPE\_var is destroyed, the copy of the TYPE is released. The From parameter specifies the var class that points to the data to be copied.

#### TYPE \*operator->(); TYPE \*operator->() const;

These operators return a pointer to the TYPE owned by the var class. The var class continues to own the TYPE and it is the responsibility of the var class to release TYPE. You cannot use the operator-> until the var owns a valid TYPE. Do not try to release this return value or access this return value after the TYPE var has been destroyed.

# TYPE\_var\_ptr in() const; TYPE\_var\_ptr&inout(); TYPE\_var\_ptr retn();

Because implicit conversions can sometimes cause a problem with some C++ compilers and with code readability, the <code>TYPE\_var</code> types also support member functions that allow them to be explicitly converted for purposes of parameter passing. To pass a <code>TYPE\_var</code> and an <code>in</code> parameter, call the <code>in()</code> member function; for <code>inout</code> parameters, the <code>inout()</code> member function; for out parameters, the <code>out()</code> member function. To obtain a return value from the <code>TYPE\_var</code>, call the <code>return()</code> function. For each <code>TYPE\_var</code> type, the return types of each of these functions will match the type shown in the following table for the <code>in</code>, <code>inout</code>, <code>out</code>, and return modes for the underlying type <code>TYPE</code>, respectively.

Some differences occur in the operators supported for the user-defined data types. The following table describes the various operators supported by each OMG IDL data type, in the generated C++ code. Because the assignment operators are supported for all of the data types described in the following table, they are not included in the comparison.



Table 13-3 Comparison of Operators Supported for User-defined Data Type var Classes

| OMG IDL Data Type | operator -> | operator []         |
|-------------------|-------------|---------------------|
| struct            | Yes         | No                  |
| union             | Yes         | No                  |
| sequence          | Yes         | Yes, non-const only |
| array             | No          | Yes                 |

The signatures are as shown in the following table.

Table 13-4 Operator Signatures for \_var Classes

| OMG IDL Data<br>Type | Operator Member Functions  |
|----------------------|--|
| struct               | TYPE * operator-> () TYPE * operator-> () const  |
| union                | <pre>TYPE * operator-&gt; () TYPE * operator-&gt; () const</pre>   |
| sequence             | <pre>TYPE * operator-&gt; () TYPE * operator-&gt; () const TYPE &amp; operator[](CORBA::Long index)</pre>      |
| array                | <pre>TYPE_slice &amp; operator[](CORBA::Long index) TYPE_slice &amp; operator[](CORBA::Long index) const</pre> |

- Sequence vars
- Array vars
- String vars

# 13.3.1 Sequence vars

Sequence vars support the following additional operator[] member function:

#### TYPE &operator[](CORBA::ULong Index);

This operator invokes the <code>operator[]</code> of sequence owned by the var class. The <code>operator[]</code> returns a reference to the appropriate element of the sequence at the specified index. The <code>Index</code> argument specifies the index of the element to return. This index cannot be greater than the current sequence length.

# 13.3.2 Array vars

Array vars do not support operator->, but do support the following additional operator[] member functions to access the array elements:

```
TYPE slice&operator[](CORBA::ULong Index); const TYPE_slice &operator[]
(CORBA::ULong Index) const;
```

These operators return a reference to the array slice at the specified index. An array slice is an array with all the dimensions of the original array except the first dimension. The member functions for the array-generated classes use a pointer to a slice to return pointers to an array.

The Index argument specifies the index of the slice to return. This index cannot be greater than the array dimension.

# 13.3.3 String vars

The String vars in the member functions described in this section and in the section Sequence vars have a TYPE of char \*. String vars support additional member functions, as follows:

#### String var(char \* str)

This constructor makes a String\_var from a string. The str argument specifies the string that will be assumed. The user must not use the str pointer to access data.

#### String var(const char \* str) String var(const String var &var)

This constructor makes a String\_var from a const string. The str argument specifies the const string that will be copied. The var argument specifies a reference to the string to be copied.

#### String var &operator=(char \* str)

This assignment operator first releases the contained string using <code>CORBA::string\_free</code>, and then assumes ownership of the input string. The <code>str</code> argument specifies the string whose ownership will be assumed by this <code>String\_var</code> object.

# String\_var &operator=(const char \* str) String\_var &operator=(const String\_var &var)

This assignment operator first releases the contained string using <code>CORBA::string\_free</code>, and then copies the input string. The <code>Data</code> argument specifies the string whose ownership will be assumed by this <code>String var</code> object.

#### char operator[] (Ulong Index) char operator[] (Ulong Index) const

These array operators are superscripting operators that provide access to characters within the string. The Index argument specifies the index of the array to use in accessing a particular character within the array. Zero-based indexing is used. The returned value of the Char operator[] (Ulong Index) function can be used as an Ivalue. The returned value of the Char operator[] (Ulong Index) const function cannot be used as an Ivalue.

#### out Classes

#### 13.3.3.1 out Classes

Structured types (struct, union, sequence), arrays, and interfaces have a corresponding generated \_out class. The out class is provided for simplifying the memory management of pointers to variable-length and fixed-length types. For more information about out classes and the common member functions, see the section Using out Classes.

Some differences occur in the operators supported for the user-defined data types. The following table describes the various operators supported by each OMG IDL data type, in the generated C++ code. Because the assignment operators are supported for all of the data types described in the following table, they are not included in the comparison.

Table 13-5 Comparison of Operators Supported for User-defined Data Type Out Classes

| OMG IDL Data Type | operator -> | operator[] |
|-------------------|-------------|------------|
| struct            | Yes         | No         |



Table 13-5 (Cont.) Comparison of Operators Supported for User-defined Data Type Out Classes

| OMG IDL Data Type | operator -> | operator[]          |
|-------------------|-------------|---------------------|
| union             | Yes         | No                  |
| sequence          | Yes         | Yes, non-const only |
| array             | No          | Yes                 |

The signatures are as shown in the following table.

Table 13-6 Operator Signatures for \_out Classes

| OMG IDL Data Type | Operator Member Functions                                       |
|-------------------|---|
| struct            | TYPE * operator-> ()  |
|                   | TYPE * operator-> () const                                      |
| union             | TYPE * operator-> ()  |
|                   | TYPE * operator-> () const                                      |
| sequence          | TYPE * operator-> ()  |
|                   | TYPE * operator-> () const                                      |
|                   | <pre>TYPE &amp; operator[](CORBA::Long index)</pre>             |
| array             | <pre>TYPE_slice &amp; operator[](CORBA::Long index)</pre>       |
|                   | <pre>TYPE_slice &amp; operator[](CORBA::Long index) const</pre> |

# 13.4 Using out Classes

When a  $TYPE\_var$  is passed as an out parameter, any previous value it referred to must be implicitly deleted. To give the ORB enough hooks to meet this requirement, each  $T\_var$  type has a corresponding TYPE out type that is used solely as the out parameter type.

#### Note:

The  $\_{\tt out}$  classes are not intended to be instantiated directly by the programmer. Specify an  $\_{\tt out}$  class only in function signatures.

The general form for TYPE out types for variable-length types is as follows:

```
// C++
class TYPE out
{
  public:
    TYPE out(TYPE*& p) : ptr (p) { ptr = 0; }
    TYPE out(TYPE var& p) : ptr (p.ptr ) { delete ptr ; ptr = 0;}
    TYPE out(TYPE out& p) : ptr_(p.ptr_) {}
    TYPE out& operator=(TYPE out& p) { ptr = p.ptr ;
                                        return *this:
     Type out& operator=(Type* p) { ptr = p; return *this; }
    operator Type*&() { return ptr ; }
     Type*& ptr() { return ptr ; }
     Type* operator->() { return ptr ; }
private:
     Type*& ptr ;
     // assignment from TYPE var not allowed
    void operator=(const TYPE var&):
};
```

The first constructor binds the reference data member with the T\*&argument and sets the pointer to the zero (0) pointer value. The second constructor binds the reference data member with the pointer held by the  $TYPE\_var$  argument, and then calls delete on the pointer (or  $string\_free()$  in the case of the  $String\_out$  type or  $TYPE\_free()$  in the case of a  $TYPE\_var$  for an array type TYPE). The third constructor, the copy constructor, binds the reference data member to the same pointer referenced by the data member of the constructor argument.

Assignment from another  $TYPE\_$ out copies the  $TYPE^*$  referenced by the  $TYPE\_$ out argument to the data member. The overloaded assignment operator for  $TYPE^*$  simply assigns the pointer argument to the data member. Note that assignment does not cause any previously held pointer to be deleted; in this regard, the  $TYPE\_$ out type behaves exactly as a  $TYPE^*$ . The  $TYPE^*$ &conversion operator returns the data member. The ptr() member function, which can be used to avoid having to rely on implicit conversion, also returns the data member. The overloaded arrow operator (operator->()) allows access to members of the data structure pointed to by the  $TYPE^*$  data member. Compliant applications may not call the overloaded operator->() unless the TYPE out has been initialized with a valid nonNULL TYPE \*.

Assignment to a  $TYPE\_out$  from instances of the corresponding  $TYPE\_var$  type is disallowed because there is no way to determine whether the application developer wants a copy to be performed, or whether the  $TYPE\_var$  must yield ownership of its managed pointer so it can be

assigned to the  $TYPE\_out$ . To perform a copy of a  $TYPE\_var$  to a  $TYPE\_out$ , the application must use new, as follows:

The in () function called on t typically returns a const TYPE&, suitable for invoking the copy constructor of the newly allocated T instance.

Alternatively, to make the  $TYPE\_var$  yield ownership of its managed pointer so it can be returned in a  $T\_out$  parameter, the application must use the  $TYPE\_var::\_retn()$  function, as follows:

```
// C++
TYPE_var t = ...;
my out = t. retn();  // t yields ownership, no copy
```

#### Note:

The <code>TYPE\_out</code> types are not intended to serve as general-purpose data types to be created and destroyed by applications; they are used only as types within operation signatures to allow necessary memory management side-effects to occur properly.

- Object Reference out Parameter
- Sequence outs
- Array outs
- String outs

# 13.4.1 Object Reference out Parameter

When a \_var is passed as an out parameter, any previous value it refers to must be implicitly released. To give C++ mapping implementations enough hooks to meet this requirement, each object reference type results in the generation of an \_out type that is used solely as the out parameter type. For example, interface <code>TYPE</code> results in the object reference type <code>TYPE\_ptr</code>, the helper type <code>TYPE\_var</code>, and the out parameter type <code>TYPE\_out</code>. The general form for object reference \_out types is as follows:

```
// C++
class TYPE_out
{
  public:
    TYPE_out(TYPE_ptr& p) : ptr_(p) { ptr_ = TYPE::_nil(); }
    TYPE_out(TYPE_var& p) : ptr_(p.ptr_) {
    release(ptr_); ptr_ = TYPE::_nil();
}
TYPE_out(TYPE_out& a) : ptr_(a.ptr_) {}
TYPE_out& operator=(TYPE_out& a) {
    ptr_ = a.ptr_; return *this;
}
```



```
TYPE_out& operator=(const TYPE_var& a) {
ptr_ = TYPE::_duplicate(TYPE_ptr(a)); return *this;
}
TYPE_out& operator=(TYPE_ptr p) { ptr_ = p; return *this; }
operator TYPE_ptr&() { return ptr_; }
TYPE_ptr& ptr() { return ptr_; }
TYPE_ptr operator->() { return ptr_; }
private:
    TYPE_ptr& ptr_;
};
```

# 13.4.2 Sequence outs

Sequence outs support the following additional operator[] member function:

```
TYPE &operator[](CORBA::ULong Index);
```

This operator invokes the <code>operator[]</code> of the sequence owned by the out class. The <code>operator[]</code> returns a reference to the appropriate element of the sequence at the specified index. The <code>Index</code> argument specifies the index of the element to return. This index cannot be greater than the current sequence length.

# 13.4.3 Array outs

Array outs do not support operator->, but do support the following additional operator[] member functions to access the array elements:

```
TYPE_slice&operator[](CORBA::ULong Index); const TYPE_slice &operator[]
(CORBA::ULong Index) const;
```

These operators return a reference to the array slice at the specified index. An array slice is an array with all the dimensions of the original array except the first dimension. The member functions for the array-generated classes use a pointer to a slice to return pointers to an array. The Index argument specifies the index of the slice to return. This index cannot be greater than the array dimension.

# 13.4.4 String outs

When a <code>String\_var</code> is passed as an out parameter, any previous value it refers to must be implicitly freed. To give C++ mapping implementations enough hooks to meet this requirement, the string type also results in the generation of a <code>String\_out</code> type in the CORBA namespace that is used solely as the string out parameter type. The general form for the <code>String\_out</code> type is as follows:

```
// C++
class String_out
{
  public:
    String_out(char*& p) : ptr_(p) { ptr_ = 0; }
    String_out(String_var& p) : ptr_(p.ptr_) {
        string_free(ptr_); ptr_ = 0;
    }
    String_out(String_out& s) : ptr_(s.ptr_) {}
    String_out& operator=(String_out& s) {
        ptr_ = s.ptr_; return *this;
    }
}
```

```
}
String_out& operator=(char* p) {
    ptr_ = p; return *this;
}
String_out& operator=(const char* p) {
    ptr_ = string_dup(p); return *this;
}
operator char*&() { return ptr_; }
char*& ptr() { return ptr_; }
private:
    char*& ptr_;

// assignment from String_var disallowed void operator=(const String_var&);
};
```

The first constructor binds the reference data member with the  ${\tt char}^*$  &argument. The second constructor binds the reference data member with the  ${\tt char}^*$  held by the  ${\tt String\_var}$  argument, and then calls  ${\tt string\_free}()$  on the string. The third constructor, the copy constructor, binds the reference data member to the same  ${\tt char}^*$  bound to the data member of its argument.

Assignment from another <code>String\_out</code> copies the <code>char\*</code> referenced by the argument <code>String\_out</code> to the <code>char\*</code> referenced by the data member. The overloaded assignment operator for <code>char\*</code> simply assigns the <code>char\*</code> argument to the data member. The overloaded assignment operator for <code>constchar\*</code> duplicates the argument and assigns the result to the data member. The assignment does not cause any previously held string to be freed; in this regard, the <code>String\_out</code> type behaves exactly as a <code>char\*</code>. The <code>char\*</code> &conversion operator returns the data member. The <code>ptr()</code> member function, which can be used to avoid having to rely on implicit conversion, also returns the data member.

Assignment from <code>String\_var</code> to a <code>String\_out</code> is disallowed because of the memory management ambiguities involved. Specifically, it is not possible to determine whether the string owned by the <code>String\_var</code> must be taken over by the <code>String\_out</code> without copying, or if it must be copied. Disallowing assignment from <code>String\_var</code> forces the application developer to make the choice explicitly, as follows:

On the line marked with the comment "1," the caller is explicitly copying the string held by the String\_var and assigning the result to the out argument. Alternatively, the caller could use the technique shown on the line marked with the comment "2" to force the String\_var to give up its ownership of the string it holds so that it may be returned in the out argument without incurring memory management errors.

# 13.5 Argument Passing Considerations

The mapping of parameter passing modes attempts to balance the need for both efficiency and simplicity. For primitive types, enumerations, and object references,

the modes are straightforward, passing the type P for primitives and enumerations and the type A ptr for an interface type A.

Aggregate types are complicated by the question of when and how parameter memory is allocated and deallocated. Mapping in parameters is straightforward because the parameter storage is caller-allocated and read-only. The mapping for out and inout parameters is more problematic. For variable-length types, the callee must allocate some if not all of the storage. For fixed-length types, such as a *Point* type

represented as a struct containing three floating point members, caller allocation is preferable (to allow stack allocation).

To accommodate both kinds of allocation, avoid the potential confusion of split allocation, and eliminate confusion with respect to when copying occurs, the mapping is  $\mathbb{T}$  & for a fixed-length aggregate  $\mathbb{T}$  and  $\mathbb{T}^*$  & for a variable-length  $\mathbb{T}$ . This approach has the unfortunate consequence that usage for structs depends on whether the struct is fixed- or variable-length; however, the mapping is consistently  $\mathbb{T}$  var & if the caller uses the managed type  $\mathbb{T}$  var.

The mapping for out and inout parameters additionally requires support for deallocating any previous variable-length data in the parameter when a  $\mathbb{T}_{var}$  is passed. Even though their initial values are not sent to the operation, the Oracle Tuxedo includes out parameters because the parameter could contain the result from a previous call. The provision of the  $\mathbb{T}_{out}$  types is intended to give implementations the hooks necessary to free the inaccessible storage while converting from the  $\mathbb{T}_{var}$  types. The following examples demonstrate the compliant behavior:

```
// IDL
struct S { string name; float age; };
void f(out S p);
// C++
S var s;
f(s);
// use s
                // first result will be freed
f(s);
S *sp;
               // need not initialize before passing to out
f(sp);
// use sp
delete sp;
              // cannot assume next call will free old value
f(sp);
```

The implicit deallocation of previous values for out and inout parameters works only with  $\mathbb{T}$  var types, not with other types:



Each call to the  ${\tt q}$  function in the loop results in a memory leak because the caller is not invoking string free on the out result. There are two ways to fix this, as shown below:

```
// C++
char *s;
String_var svar;
for (int i = 0 ; i < 10; i++) {
    q(s);
    string_free(s); // explicit deallocation
    // OR:
    q(svar); // implicit deallocation
}</pre>
```

Using a plain <code>char\*</code> for the <code>out</code> parameter means that the caller must explicitly deallocate its memory before each reuse of the variable as an <code>out</code> parameter, while using a <code>String\_var</code> means that any deallocation is performed implicitly upon each use of the variable as an <code>out</code> parameter.

Variable-length data must be explicitly released before being overwritten. For example, before assigning to an inout string parameter, the implementor of an operation may first delete the old character data. Similarly, an inout interface parameter must be released before being reassigned. One way to ensure that the parameter storage is released is to assign it to a local T var variable with an automatic release, as in the following example:

```
// IDL
interface A;
void f(inout string s, inout A obj);
// C++
void Aimpl::f(char *&s, A_ptr &obj) {
    String_var s_tmp = s;
    s = /* new data */;
    A_var obj_tmp = obj;
    obj = /* new reference */
}
```

For parameters that are passed or returned as a pointer  $(\mathbb{T}^*)$  or as a reference to a pointer  $(\mathbb{T}^*)$ , an application is not allowed to pass or return a NULL pointer; the result of doing so is undefined. In particular, a caller may not pass a NULL pointer under any of the following circumstances:

- in and inout string
- in and inout array (pointer to first element)

However, a caller may pass a reference to a pointer with a NULL value for out parameters, because the callee does not examine the value, but overwrites it. A callee may not return a NULL pointer under any of the following circumstances:

- out and return variable-length struct
- out and return variable-length union
- out and return string
- out and return sequence
- out and return variable-length array, return fixed-length array



- out and return any
- Operation Parameters and Signatures

# 13.5.1 Operation Parameters and Signatures

The following table displays the mapping for the basic OMG IDL parameter passing modes and return type according to the type being passed or returned. The following table displays the same information for  $\mathbb{T}_{\mathtt{var}}$  types. The following table is merely for informational purposes; it is expected that operation signatures for both clients and servers will be written in terms of the parameter-passing modes shown in the following table, with the exception that the  $\mathbb{T}_{\mathtt{out}}$  types will be used as the actual parameter types for all  $\mathtt{out}$  parameters.

It is also expected that  $\mathbb{T}_{var}$  types will support the necessary conversion operators to allow them to be passed directly. Callers must always pass instances of either  $\mathbb{T}_{var}$  types or the base types shown in the following table, and callees must treat their  $\mathbb{T}_{out}$  parameters as if they were actually the corresponding underlying types shown in the following table.

In following table, fixed-length arrays are the only case where the type of an out

parameter differs from a return value, which is necessary because C++ does not allow a function to return an array. The mapping returns a pointer to a *slice* of the

array, where a slice is an array with all the dimensions of the original array specified except the first dimension.

Table 13-7 Basic Argument and Result Passing

| Data Type                              | In              | Inout       | Out            | Return                          |
|--|-----------------|-------------|----------------|---------------------------------|
| short                                  | Short           | Short&      | Short&         | Short                           |
| long                                   | Long            | Long&       | Long&          | Long&                           |
| unsigned short                         | UShort          | UShort&     | UShort&        | UShort&                         |
| unsigned long                          | ULong           | ULong&      | ULong&         | ULong&                          |
| float                                  | Float           | Float&      | Float&         | Float&                          |
| double                                 | Double          | Double&     | Double&        | Double&                         |
| boolean                                | Boolean         | Boolean&    | Boolean&       | Boolean&                        |
| char                                   | Char            | Char&       | Char&          | Char&                           |
| wchar                                  | WChar           | WChar&      | WChar&         | WChar&                          |
| octet                                  | Octet           | Octet&      | Octet&         | Octet&                          |
| enum                                   | enum            | enum&       | enum&          | enum&                           |
| object reference ptr (See Note below.) | objref_ptr      | objref_ptr& | objref_ptr&    | objref_ptr                      |
| struct, fixed                          | const struct&   | struct&     | struct&        | struct                          |
| struct, variable                       | const struct&   | struct&     | struct*&       | struct*                         |
| union, fixed                           | const union&    | union&      | union&         | union                           |
| union, variable                        | const union*    | union&      | union&         | union                           |
| string                                 | const char      | char*&      | char*&         | char*                           |
| wstring                                | const WChar     | WChar*&     | WChar*&        | WChar*                          |
| sequence                               | const sequence& | sequence&   | sequence*&     | sequence*                       |
| array, fixed                           | const array     | array       | array          | array slice * (See Note below.) |
| array, variable                        | const array     | array       | array slice *& | array slice *                   |



Table 13-7 (Cont.) Basic Argument and Result Passing

| Data Type | In         | Inout | Out   | Return |  |
|-----------|------------|-------|-------|--------|--|
| any       | const any& | any&  | any*& | any*   |  |



The Object reference ptr data type includes pseudo-object references. The array slice return is an array with all the dimensions of the original array except the first dimension.

A caller is responsible for providing storage for all arguments passed as in arguments.

Table 13-8 T\_var Argument and Result Passing

| Data Type                              | In                     | Inout         | Out           | Return       |
|--|------------------------|---------------|---------------|--------------|
| object reference var (See Note below.) | const object           | objref_var&   | objref_var&   | objref_var   |
| struct_var                             | const struct_var&      | struct_var&   | struct_var&   | struct_var   |
| union_var                              | const union_var&       | union_var&    | union_var&    | union_var    |
| string_var                             | const string_var&      | string_var&   | string_var&   | string_var   |
| sequence_var                           | const<br>sequence_var& | sequence_var& | sequence_var& | sequence_var |
| array_var                              | const array_var&       | array_var&    | array_var&    | array_var    |
| any_var                                | const any_var&         | any_var&      | any_var&      | any_var      |

Note:

The object reference var data type includes pseudo-object references.

The following table and table describe the caller's responsibility for storage associated with inout and out parameters and for return results.

**Table 13-9 Caller Argument Storage Responsibilities** 

| Туре           | Inout Param | Out Param | Return Result |
|----------------|-------------|-----------|---------------|
| short          | 1           | 1         | 1             |
| long           | 1           | 1         | 1             |
| unsigned short | 1           | 1         | 1             |
| unsigned long  | 1           | 1         | 1             |
| float          | 1           | 1         | 1             |
| double         | 1           | 1         | 1             |
| boolean        | 1           | 1         | 1             |
| char           | 1           | 1         | 1             |
| wchar          | 1           | 1         | 1             |
| octet          | 1           | 1         | 1             |

Table 13-9 (Cont.) Caller Argument Storage Responsibilities

| Туре                 | Inout Param | Out Param | Return Result |
|----------------------|-------------|-----------|---------------|
| enum                 | 1           | 1         | 1             |
| object reference ptr | 2           | 2         | 2             |
| struct, fixed        | 1           | 1         | 1             |
| struct, variable     | 1           | 3         | 3             |
| union, fixed         | 1           | 1         | 1             |
| union, variable      | 1           | 3         | 3             |
| string               | 4           | 3         | 3             |
| wstring              | 4           | 3         | 3             |
| sequence             | 5           | 3         | 3             |
| array, fixed         | 1           | 1         | 6             |
| array, variable      | 1           | 6         | 6             |
| any                  | 5           | 3         | 3             |

In particular, exception used as a data type and a function name.

Table 13-10 Argument Passing Cases

| Case |   |
|------|---|
| 1    | Caller allocates all necessary storage, except that which may be encapsulated and managed within the parameter itself. For inout parameters, the caller provides the initial value, and the callee may change that value. For out parameters, the caller allocates the storage but need not initialize it, and the callee sets the value. Function returns are by value.  |
| 2    | Caller allocates storage for the object reference. For inout parameters, the caller provides an initial value; if the callee wants to reassign the inout parameter, it will first call CORBA::release on the original input value. To continue to use an object reference passed in as an inout, the caller must first duplicate the reference. The caller is responsible for the release of all out and return object references. Release of all object references embedded in other structures is performed automatically by the structures themselves.   |
| 3    | For out parameters, the caller allocates a pointer and passes it by reference to the callee. The callee sets the pointer to point to a valid instance of the parameter's type. For returns, the callee returns a similar pointer. The callee is not allowed to return a NULL pointer in either case.  In both cases, the caller is responsible for releasing the returned storage. To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the caller or is located in a different address space. Following the completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned instance into a new instance, and modify the new instance. |
| 4    | For inout strings, the caller provides storage for both the input string and the <code>char*</code> pointing to it. Since the callee may deallocate the input string and reassign the <code>char*</code> to point to new storage to hold the output value, the caller should allocate the input string using <code>string_alloc()</code> . The size of the out string is, therefore, not limited by the size of the in string. The caller is responsible for deleting the storage for the out using <code>string_free()</code> . The callee is not allowed to return a <code>NULL</code> pointer for an inout, out, or return value.  |

Table 13-10 (Cont.) Argument Passing Cases

| Case |   |
|------|---|
| 5    | For inout sequences and anys, assignment or modification of the sequence or any may cause deallocation of owned storage before any reallocation occurs, depending upon the state of the Boolean release parameter with which the sequence or any was constructed.   |
| 6    | For out parameters, the caller allocates a pointer to an array slice, which has all the same dimensions of the original array except the first, and passes the pointer by reference to the callee. The callee sets the pointer to point to a valid instance of the array.  For returns, the callee returns a similar pointer. The callee is not allowed to return a NULL pointer in either case. In both cases, the caller is responsible for releasing the returned storage. |
|      | To maintain local/remote transparency, the caller must always release the returned storage, regardless of whether the callee is located in the same address space as the callee or is located in a different address space. Following completion of a request, the caller is not allowed to modify any values in the returned storage—to do so, the caller must first copy the returned array instance into a new array instance, and modify the new instance.                |



# **CORBA API**

This chapter describes the Oracle Tuxedo implementation of the CORBA core member functions in C++ and their extensions. It also describes pseudo-objects and their relationship to C++ classes. Pseudo-objects are object references that cannot be transmitted across the network. Pseudo-objects are similar to other objects; however, because the ORB owns them, they cannot be extended.

#### Note:

Some of the information in this chapter is taken from the *Common Object Request Broker: Architecture and Specification*. Revision 2.4.2, February 2001, published by the Object Management Group (OMG). Used with permission of the OMG. The Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported in Tuxedo 9.x. All Oracle Tuxedo CORBA Java client and Oracle Tuxedo CORBA Java client ORB text references, associated code samples, etc. must only be used:

- to help implement/run third party Java ORB libraries, and
- for programmer reference only.

Technical support for third party CORBA Java ORBs must be provided by their respective vendors. Oracle Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

- Global Classes
- Pseudo-objects
- Any Class Member Functions
- Context Member Functions
- ContextList Member Functions
- NamedValue Member Functions
- NVList Member Functions
- Object Member Functions
- CORBA Member Functions
- ORB Member Functions
- ORB Initialization Member Function
- Policy Member Functions
- PortableServer Member Functions
- POA Current Member Functions
- POAManager Member Functions
- POA Policy Member Objects



- Request Member Functions
- Strings
- Wide Strings
- TypeCode Member Functions
- Exception Member Functions
- Standard Exceptions
- ExceptionList Member Functions

# 14.1 Global Classes

The following Oracle Tuxedo classes are global in scope:

- CORBA
- Tobj

These classes contain the predefined types, classes, and functions used in Oracle Tuxedo development.

The CORBA class contains the classes, data types, and member functions essential to using an Object Request Broker (ORB) as defined by CORBA. The Oracle Tuxedo extensions to CORBA are contained in the Tobj C++ class. The Tobj class contains data types, nested classes, and member functions that Oracle Tuxedo provides as an extension to CORBA.

Using CORBA data types and member functions in the Oracle Tuxedo product requires the CORBA:: prefix. For example, a Long is a CORBA::Long. Likewise, to use Tobj nested classes and member functions in the Oracle Tuxedo product, you need the Tobj::prefix. For example, FactoryFinder is Tobj::FactoryFinder.

# 14.2 Pseudo-objects

Pseudo-objects are represented as local classes, which reside in the CORBA class. A pseudo-object and its corresponding member functions are named using a nested class structure. For example, an ORB object is a CORBA::ORB and a Current object is a CORBA::Current.

# 14.3 Any Class Member Functions

This section describes the member functions of the Any class.

The mapping of these member functions to C++ is as follows:

```
class CORBA
{
  class Any
{
  public:
  Any ();
  Any (const Any&);
  Any (TypeCode_ptr tc, void *value, Boolean release =
  CORBA_ FALSE);
  ~Any ();
  Any & operator=(const Any&);
```



```
void operator<<=(Short);</pre>
void operator<<=(UShort);</pre>
void operator<<=(Long);</pre>
void operator<<=(ULong);</pre>
void operator<<=(Float);</pre>
void operator<<=(Double);</pre>
void operator<<=(const Any&);</pre>
void operator<<=(const char*);</pre>
void operator<<=(Object ptr);</pre>
void operator<<=(from boolean);</pre>
void operator<<=(from char);</pre>
void operator<<=(from octet);</pre>
void operator<<=(from string);</pre>
Boolean operator>>=(Short&) const;
Boolean operator>>=(UShort&) const;
Boolean operator>>=(Long&) const;
Boolean operator>>=(ULong&) const;
Boolean operator>>=(Float&) const;
Boolean operator>>=(Double&) const;
Boolean operator>>=(Any&) const;
Boolean operator>>=(char*&) const;
Boolean operator>>=(Object ptr&) const;
Boolean operator>>=(to boolean) const;
Boolean operator>>=(to char) const;
Boolean operator>>=(to octet) const;
Boolean operator>>=(to object) const;
Boolean operator>>=(to string) const;
TypeCode ptr type()const;
void replace(TypeCode ptr, void *, Boolean);
void replace(TypeCode ptr, void *);
const void * value() const;
};
}; //CORBA
   CORBA::Any::Any()
   CORBA::Any::Any(const CORBA::Any & InitAny)
   CORBA::Any::Any(TypeCode ptr TC, void * Value, Boolean Release)
   CORBA::Any::~Any()
   CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)
   void CORBA::any::operator<<=()</pre>
   CORBA::Boolean CORBA::Any::operator>>=()
   CORBA::Any::operator<<=()
   CORBA::Boolean CORBA::Any::operator>>=()
   CORBA::TypeCode ptr CORBA::Any::type() const
   void CORBA::Any::replace()
```

# 14.3.1 CORBA::Any::Any()

Synopsis

- C++ Binding
- Arguments
- Description
- Return Values

#### 14.3.1.1 Synopsis

Constructs the Any object.

# 14.3.1.2 C++ Binding

CORBA::Any::Any()

#### 14.3.1.3 Arguments

None.

### 14.3.1.4 Description

This is the default constructor for the CORBA::Any class. It creates an Any object with a TypeCode of type tc\_null and a value of 0 (zero).

#### 14.3.1.5 Return Values

None.

# 14.3.2 CORBA::Any::Any(const CORBA::Any & InitAny)

- Synopsis
- · C++ Binding
- Argument
- Description
- Return Values

# 14.3.2.1 Synopsis

Constructs the Any object that is a copy of another Any object.

# 14.3.2.2 C++ Binding

CORBA::Any::Any(const CORBA::Any & InitAny)

# 14.3.2.3 Argument

InitAny

Refers to the CORBA:: Any to copy.

#### 14.3.2.4 Description

This is the copy constructor for the CORBA: :Any class. This constructor duplicates the TypeCode reference of the Any that is passed in.

The type of copying to be performed is determined by the release flag of the Any object to be copied. If release evaluates as CORBA\_TRUE, the constructor deep-copies the parameter's value; if release evaluates as CORBA\_FALSE, the constructor shallow-copies the parameter's value. Using a shallow copy gives you more control to optimize memory allocation, but the caller must ensure the Any does not use memory that has been freed.

#### 14.3.2.5 Return Values

None.

# 14.3.3 CORBA::Any::Any(TypeCode ptr TC, void \* Value, Boolean Release)

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.3.3.1 Synopsis

Creates the Any object using a TypeCode and a value.

### 14.3.3.2 C++ Binding

CORBA::Any::Any(TypeCode ptr TC, void \* Value, Boolean Release)

# 14.3.3.3 Arguments

TC

A pointer to a TypeCode pseudo-object reference, specifying the type to be created.

#### Value

A pointer to the data to be used to create the Any object. The data type of this argument must match the TypeCode specified.

#### Release

Determines whether the Any assumes ownership of the memory specified by the Value argument. If Release is CORBA\_TRUE, the Any assumes ownership. If Release is CORBA\_FALSE, the Any does not assume ownership; the data pointed to by the Value argument is not released upon assignment or destruction.

# 14.3.3.4 Description

This constructor is used with the nontype-safe Any interface. It duplicates the specified TypeCode object reference and then inserts the data pointed to by value inside the

Any object.

#### 14.3.3.5 Return Values

None.

# 14.3.4 CORBA::Any::~Any()

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.3.4.1 Synopsis

Destructor for the Any.

# 14.3.4.2 C++ Binding

CORBA::Any::~Any()

### 14.3.4.3 Arguments

None.

# 14.3.4.4 Description

This destructor frees the memory that the CORBA::Any holds (if the Release flag is specified as CORBA TRUE), and releases the TypeCode pseudo-object reference contained in the Any.

#### 14.3.4.5 Return Values

None.

# 14.3.5 CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.3.5.1 Synopsis

Any assignment operator.

# 14.3.5.2 C++ Binding

```
CORBA::Any & CORBA::Any::operator=(const CORBA::Any & InitAny)
```

#### 14.3.5.3 Arguments

#### InitAny

A reference to an Any to use in the assignment. The Any to use in the assignment determines whether the Any assumes ownership of the memory in Value. If Release is CORBA\_TRUE, the Any assumes ownership and deep-copies the InitAny argument's value; if Release is CORBA\_FALSE, the Any shallow-copies the InitAny argument's value.

### 14.3.5.4 Description

This is the assignment operator for the <code>Any</code> class. Memory management of this member function is determined by the current value of the <code>Release</code> flag. The current value of the <code>Release</code> flag determines whether the current memory is released before the assignment. If the current <code>Release</code> flag is <code>CORBA\_TRUE</code>, the <code>Any</code> releases any value previously held; if the current <code>Release</code> flag is <code>CORBA\_FALSE</code>, the <code>Any</code> does not release any value previously held.

#### 14.3.5.5 Return Values

Returns the Any, which holds the copy of the InitAny.

# 14.3.6 void CORBA::any::operator<<=()

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

# 14.3.6.1 Synopsis

Type safe Any insertion operators.

# 14.3.6.2 C++ Binding

```
void CORBA::Any::operator<<=(CORBA::Short Value)
void CORBA::Any::operator<<=(CORBA::UShort Value)
void CORBA::Any::operator<<=(CORBA::Long Value)
void CORBA::Any::operator<<=(CORBA::Ulong Value)
void CORBA::Any::operator<<=(CORBA::Float Value)
void CORBA::Any::operator<<=(CORBA::Double Value)
void CORBA::Any::operator<<=(const CORBA::Any & Value)
void CORBA::Any::operator<<=(const char * Value)
void CORBA::Any::operator<<=(CORBA::Any & Value)</pre>
```

### 14.3.6.3 Argument

#### Value

Type specific value to be inserted into the Any.

#### 14.3.6.4 Description

This insertion member function performs type-safe insertions. If the Any had a previous value, and the Release flag is CORBA\_TRUE, the memory is deallocated and the previous TypeCode object reference is freed. The new value is inserted into the Any by copying the value passed in using the Value parameter. The appropriate TypeCode reference is duplicated.

#### 14.3.6.5 Return Values

None.

# 14.3.7 CORBA::Boolean CORBA::Any::operator>>=()

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

### 14.3.7.1 Synopsis

Type safe Any extraction operators.

### 14.3.7.2 C++ Binding

```
CORBA::Boolean CORBA::Any::operator>>=(
CORBA::Short & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CORBA::UShort & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CORBA::Long & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CORBA::Ulong & Value) const

CORBA::Ulong & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CORBA::Float & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CORBA::Double & Value) const

CORBA::Boolean CORBA::Any::operator>>=(
CO
```

### 14.3.7.3 Argument

The Value argument is a reference to the relevant object that receives the output of the value contained in the Any object.

### 14.3.7.4 Description

This extraction member function performs type-safe extractions. If the <code>Any</code> object contains the specified type, this member function assigns the pointer of the <code>Any</code> to the output reference value, <code>Value</code>, and <code>CORBA\_TRUE</code> is returned. If the <code>Any</code> does not contain the appropriate type, <code>CORBA\_FALSE</code> is returned. The caller must not attempt to release or delete the storage because it is owned and managed by the <code>Any</code> object. The <code>Value</code> argument is a reference to the relevant object that receives the output of the value contained in the <code>Any</code> object. If the <code>Any</code> object does not contain the appropriate type, the value remains unchanged.

#### 14.3.7.5 Return Values

CORBA\_TRUE if the Any contained a value of the specific type. CORBA\_FALSE if the Any did not contain a value of the specific type.

# 14.3.8 CORBA::Any::operator<<=()

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

### 14.3.8.1 Synopsis

Type safe insertion operators for Any.

### 14.3.8.2 C++ Binding

```
void CORBA::Any::operator<<=(from_boolean Value)
void CORBA::Any::operator<<=(from_char Value)
void CORBA::Any::operator<<=(from_octet Value)
void CORBA::Any::operator<<=(from_string Value)</pre>
```

### 14.3.8.3 Argument

#### Value

A relevant object that contains the value to insert into the Any.

### 14.3.8.4 Description

These insertion member functions perform a type-safe insertion of a CORBA::Boolean, a CORBA::Char, or a CORBA::Octet reference into an Any. If the Any had a previous value, and its Release flag is CORBA TRUE, the memory is deallocated and the previous TypeCode object

reference is freed. The new value is inserted into the Any object by copying the value passed in using the Value parameter. The appropriate TypeCode reference is duplicated.

#### 14.3.8.5 Return Values

None.

# 14.3.9 CORBA::Boolean CORBA::Any::operator>>=()

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

### 14.3.9.1 Synopsis

Type-safe extraction operators for Any.

#### 14.3.9.2 C++ Binding

```
CORBA::Boolean CORBA::Any::operator>>=(to_boolean Value) const CORBA::Boolean CORBA::Any::operator>>=(to_char Value) const CORBA::Boolean CORBA::Any::operator>>=(to_octet Value) const CORBA::Boolean CORBA::Any::operator>>=(to_object Value) const CORBA::Boolean CORBA::Any::operator>>=(to_string Value) const CORBA::Boolean CORBA::Any::operator>>=(to_string Value) const
```

### 14.3.9.3 Argument

#### Value

A reference to the relevant object that receives the output of the value contained in the Any object. If the Any object does not contain the appropriate type, the value remains unchanged.

# 14.3.9.4 Description

These extraction member functions perform a type-safe extraction of a CORBA::Boolean, a CORBA::Char, a CORBA::Octet, a CORBA::Object, or a String reference from an Any. These member functions are helpers nested in the Any class. Their purpose is to distinguish extractions of the OMG IDL types: Boolean, char, and octet (C++ does not require these to be distinct types).

#### 14.3.9.5 Return Values

If the Any contains the specified type, this member function assigns the value in the Any object reference to the output variable, Value, and returns CORBA\_TRUE. If the Any object does not contain the appropriate type, CORBA\_FALSE is returned.

# 14.3.10 CORBA::TypeCode\_ptr CORBA::Any::type() const

Synopsis

- C++ Binding
- Arguments
- Description
- Return Values

#### 14.3.10.1 Synopsis

TypeCode accessor for Any.

### 14.3.10.2 C++ Binding

```
CORBA::TypeCode ptr CORBA::Any::type();
```

# 14.3.10.3 Arguments

None.

### 14.3.10.4 Description

This function returns the  $\protect\operatorname{TypeCode\_ptr}$  pseudo-object reference of the TypeCode object associated with the  $\protect\operatorname{Any}$ . The  $\protect\operatorname{TypeCode\_ptr}$  pseudo-object reference must be released by the  $\protect\operatorname{CORBA}$ ::release member function or must be assigned to a  $\protect\operatorname{TypeCode\_var}$  to be automatically released.

# 14.3.10.5 Return Values

TypeCode ptr contained in the Any.

# 14.3.11 void CORBA::Any::replace()

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.3.11.1 Synopsis

Nontype safe Any "insertion."

### 14.3.11.2 C++ Binding

```
void CORBA::Any::replace(TypeCode_ptr TC, void * Value,
Boolean Release = CORBA FALSE);
```

# 14.3.11.3 Arguments

#### TC

A TypeCode pseudo-object reference specifying the TypeCode value for the replaced Any object. This argument is duplicated.

#### Value

A void pointer specifying the storage pointed to by the Any object.

#### Release

Determines whether the  $\mathtt{Any}$  manages the specified  $\mathtt{Value}$  argument. If  $\mathtt{Release}$  is  $\mathtt{CORBA\_TRUE}$ , the Any assumes ownership. If  $\mathtt{Release}$  is  $\mathtt{CORBA\_FALSE}$ , the  $\mathtt{Any}$  does not assume ownership and the data pointed to by the  $\mathtt{Value}$  parameter is not released upon assignment or destruction.

### 14.3.11.4 Description

These member functions replace the data and TypeCode value currently contained in the Any with the value of the TC and Value arguments passed in. The functions perform a nontype-safe replacement, which means that the caller is responsible for consistency between the TypeCode value and the data type of the storage pointed to by the Value argument.

If the value of **Release** is CORBA\_TRUE, this function releases the existing TypeCode pseudoobject in the Any object and frees the storage pointed to be the Any object reference.

#### 14.3.11.5 Return Values

None.

# 14.4 Context Member Functions

A Context supplies optional context information associated with a method invocation.

The mapping of these member functions to C++ is as follows:

```
class CORBA
{
  class Context
{
   public:
    const char *context_name() const;
   Context_ptr parent() const;

  void create_child(const char *, Context_out);

  void set_one_value(const char *, const Any &);
  void set_values(NVList_ptr);
  void delete_values(const char *);
  void get_values(
    const char *,
  Flags,
    const char *,
```

```
NVList_out
);
```

Memory Management

# 14.4.1 Memory Management

Context has the following special memory management rule:

• Ownership of the return values of the <code>context\_name</code> and <code>parent</code> functions is maintained by the Context; these return values must not be freed by the caller.

This section describes Context member functions.

- CORBA::Context::context\_name
- CORBA::Context::create\_child
- CORBA::Context::delete\_values
- CORBA::Context::get\_values
- CORBA::Context::parent
- CORBA::Context::set\_one\_value
- CORBA::Context::set\_values

# 14.4.1.1 CORBA::Context::context\_name

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

#### 14.4.1.1.1 Synopsis

Returns the name of a given Context object.

### 14.4.1.1.2 C++ Binding

```
Const char * CORBA::Context::context name () const;
```

#### 14.4.1.1.3 Arguments

None.

#### 14.4.1.1.4 Description

This member function returns the name of a given Context object. The Context object reference owns the memory for the returned char \*. Users must not modify this memory.

#### 14.4.1.1.5 Return Values

If the member function succeeds, it returns the name of the Context object. The value may be empty if the Context object is not a child Context created by a call to CORBA::Context::create child.

If the Context object has no name, this is an empty string.

### 14.4.1.2 CORBA::Context::create child

- Synopsis
- C++ Binding
- Arguments
- Exception
- Description
- Return Values
- See Also

#### 14.4.1.2.1 Synopsis

Creates a child of the Context object.

#### 14.4.1.2.2 C++ Binding

```
void CORBA::Context::create_child (
const char * CtxName,
CORBA::Context out CtxObject);
```

### 14.4.1.2.3 Arguments

#### CtxName

The name to be associated with the child of the Context reference.

#### CtxObject

The newly created Context object reference.

#### 14.4.1.2.4 Exception

```
CORBA::NO_MEMORY
```

### 14.4.1.2.5 Description

This member function creates a child of the Context object. That is, searches on the child Context object will look for matching property names in the parent context (and so on, up the context tree), if necessary.

#### 14.4.1.2.6 Return Values

None.

#### 14.4.1.2.7 See Also

```
CORBA::ORB::get_default_context
CORBA::release
```

### 14.4.1.3 CORBA::Context::delete values

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

#### 14.4.1.3.1 Synopsis

Deletes the values for a specified attribute in the Context object.

#### 14.4.1.3.2 C++ Binding

```
void CORBA::Context::delete_values (
const char * AttrName);
```

### 14.4.1.3.3 Argument

#### AttrName

The name of the attribute whose values are to be deleted. If this argument has a trailing wildcard character (\*), all names that match the string preceding the wildcard character are deleted.

### 14.4.1.3.4 Exceptions

```
CORBA::BAD_PARAM if attribute is an empty string.

CORBA::BAD CONTEXT if no matching attributes to be deleted were found.
```

### 14.4.1.3.5 Description

This member function deletes named values for an attribute in the Context object. Note that it does not recursively do the same to its parents, if any.

#### 14.4.1.3.6 Return Values

None.

#### 14.4.1.3.7 See Also

```
CORBA::Context::create_child
CORBA::ORB::get default context
```

#### 14.4.1.4 CORBA::Context::get values

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- See Also

#### 14.4.1.4.1 Synopsis

Retrieves the values for a given attribute in the Context object within the specified scope.

#### 14.4.1.4.2 C++ Binding

```
void CORBA::Context::get_values (
const char * StartScope,
CORBA::Flags OpFlags,
const char * AttrName,
CORBA::NVList out AttrValues);
```

### 14.4.1.4.3 Arguments

#### StartScope

The Context object level at which to initiate the search for specified properties. The level is the name of the context, or parent, at which the search is started. If the value is 0 (zero), the search begins with the current Context object.

#### OpFlags

The only valid operation flag is <code>CORBA::CTX\_RESTRICT\_SCOPE</code>. If you specify this flag, the object implementation restricts the property search to the current scope only (that is, the property search is not executed recursively up the chain of the parent context); otherwise, the search continues to a wider scope until a match has been found or until all wider levels have been searched.

#### AttrName

The name of the attribute whose values are to be returned. If this argument has a trailing wildcard character (\*), all names that match the string preceding the wildcard character are returned.

#### AttrValues

Receives the values for the specified attributes (returns an NVList object) where each item in the list is a NamedValue.

#### 14.4.1.4.4 Exceptions

CORBA::BAD PARAM if attribute is an empty string.

CORBA::BAD CONTEXT if no matching attributes were found.

CORBA:: NO MEMORY if dynamic memory allocation failed.

#### 14.4.1.4.5 Description

This member function retrieves the values for a specified attribute in the Context object. These values are returned as an NVList object, which must be freed when no longer needed using the CORBA::release member function.

#### 14.4.1.4.6 Return Values

None.

#### 14.4.1.4.7 See Also

```
CORBA::Context::create_child
CORBA::ORB::get default context
```

# 14.4.1.5 CORBA::Context::parent

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.4.1.5.1 Synopsis

Returns the parent context of the Context object.

### 14.4.1.5.2 C++ Binding

```
CORBA::Context ptr CORBA::Context::parent () const;
```

#### 14.4.1.5.3 Arguments

None.

# 14.4.1.5.4 Description

This member function returns the parent context of the Context object. The parent of the Context object is an attribute owned by the Context and must not be modified or freed by the caller. This parent is nil unless the Context object was created using the CORBA::Context::create child member function.



#### 14.4.1.5.5 Return Values

If the member function succeeds, the parent context of the Context object is returned. The parent context may be nil. Use the <code>CORBA::is\_nil</code> member function to test for a nil object reference.

If the member function does not succeed, an exception is thrown. Use the CORBA::is\_nil member function to test for a nil object reference.

### 14.4.1.6 CORBA::Context::set one value

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- See Also

#### 14.4.1.6.1 Synopsis

Sets the value for a given attribute in the Context object.

#### 14.4.1.6.2 C++ Binding

```
void CORBA::Context::set_one_value (
const char * AttrName,
const CORBA::Any & AttrValue);
```

### 14.4.1.6.3 Arguments

#### AttrName

The name of the attribute to set.

#### AttrValue

The value of the attribute. Currently, the Oracle Tuxedo system supports only the string type; therefore, this parameter must contain a CORBA:: Any object with a string inside.

#### 14.4.1.6.4 Exceptions

CORBA::BAD\_PARAM if AttrName is an empty string or AttrValue does not contain a string type.

CORBA::NO MEMORY if dynamic memory allocation failed.

### 14.4.1.6.5 Description

This member function sets the value for a given attribute in the Context object. Currently, only string values are supported by the Context object. If the Context object already has an attribute with the given name, it is deleted first.

#### 14.4.1.6.6 Return Values

None.

#### 14.4.1.6.7 See Also

```
CORBA::Context::get_values
CORBA::Context::set values
```

### 14.4.1.7 CORBA::Context::set values

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

#### 14.4.1.7.1 Synopsis

Sets the values for given attributes in the Context object.

### 14.4.1.7.2 C++ Binding

```
void CORBA::Context::set_values (
CORBA::NVList ptr AttrValue);
```

### 14.4.1.7.3 Argument

#### AttrValues

The name and value of the attribute. Currently the Oracle Tuxedo system supports only the string type; therefore, all NamedValue objects in the list must have CORBA::Any objects with a string inside.

### 14.4.1.7.4 Exceptions

CORBA::BAD PARAM if any of the attribute values has a value that is not a string type.

CORBA:: NO MEMORY if dynamic memory allocation failed.

# 14.4.1.7.5 Description

This member function sets the values for given attributes in the Context object. The CORBA::NVList member function contains the property name and value pairs to be set.

#### 14.4.1.7.6 Return Values

None.

#### 14.4.1.7.7 See Also

```
CORBA::Context::get_values
CORBA::Context::set one value
```

# 14.5 ContextList Member Functions

The ContextList allows a client or server application to provide a list of context strings that must be supplied with Request invocation. For a description of the Request member functions, see the section Request Member Functions.

The ContextList differs from the Context in that the former supplies only the context strings whose values are to be looked up and sent with the request invocation (if applicable), while the latter is where those values are obtained. For a description of the Context member functions, see the section Context Member Functions.

The mapping of these member functions to C++ is as follows:

```
class CORBA
{
  class ContextList
{
  public:
  Ulong count ();
  void add(const char* ctxt);
  void add_consume(char* ctxt);
  const char* item(Ulong index);
  Status remove(Ulong index);
}; // ContextList
}// CORBA
```

- CORBA::ContextList:: count
- CORBA::ContextList::add
- CORBA::ContextList::add consume
- CORBA::ContextList::item
- CORBA::ContextList::remove

### 14.5.1 CORBA::ContextList:: count

- Synopsis
- C++ Binding
- Arguments
- Exception
- Description
- Return Values
- See Also



### 14.5.1.1 Synopsis

Retrieves the current number of items in the list.

### 14.5.1.2 C++ Binding

```
Ulong count ();
```

#### 14.5.1.3 Arguments

None.

### 14.5.1.4 Exception

If the function does not succeed, an exception is thrown.

### 14.5.1.5 Description

This member function retrieves the current number of items in the list.

#### 14.5.1.6 Return Values

If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no ContextList objects have been added, this function returns 0 (zero).

#### 14.5.1.7 See Also

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::item
CORBA::ContextList::remove
```

### 14.5.2 CORBA::ContextList::add

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Values
- See Also

# 14.5.2.1 Synopsis

Constructs a ContextList object with an unnamed item, setting only the flags attribute.

### 14.5.2.2 C++ Binding

void add(const char\* ctxt);

### 14.5.2.3 Argument

#### ctxt

Defines the memory location referred to by char\*.

### 14.5.2.4 Exception

If the member function does not succeed, a  ${\tt CORBA::NO\_MEMORY}$  exception is thrown.

### 14.5.2.5 Description

This member function constructs a ContextList object with an unnamed item, setting only the flags attribute.

The ContextList object grows dynamically; your application does not need to track its size.

#### 14.5.2.6 Return Values

If the function succeeds, the return value is a pointer to the newly created ContextList object.

#### 14.5.2.7 See Also

```
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::item
CORBA::ContextList::remove
```

# 14.5.3 CORBA::ContextList::add\_consume

- Synopsis
- · C++ Binding
- Argument
- Exception
- Description
- Return Values
- See Also

# 14.5.3.1 Synopsis

Constructs a ContextList object.

### 14.5.3.2 C++ Binding

```
void add consume(const char* ctxt);
```

# 14.5.3.3 Argument

#### ctxt

Defines the memory location referred to by char\*.

### 14.5.3.4 Exception

If the member function does not succeed, an exception is raised.

### 14.5.3.5 Description

This member function constructs a ContextList object.

The ContextList object grows dynamically; your application does not need to track its size.

#### 14.5.3.6 Return Values

If the function succeeds, the return value is a pointer to the newly created ContextList object.

#### 14.5.3.7 See Also

```
CORBA::ContextList::add
CORBA::ContextList::count
CORBA::ContextList::item
CORBA::ContextList::remove
```

# 14.5.4 CORBA::ContextList::item

- Synopsis
- · C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

# 14.5.4.1 Synopsis

Retrieves a pointer to the ContextList object, based on the index passed in.

# 14.5.4.2 C++ Binding

```
const char* item(ULong index);
```

### 14.5.4.3 Argument

#### inde

The index into the ContextList object. The indexing is zero-based.

### 14.5.4.4 Exceptions

If this function does not succeed, the  ${\tt BAD\_PARAM}$  exception is thrown.

## 14.5.4.5 Description

This member function retrieves a pointer to a ContextList object, based on the index passed in. The function uses zero-based indexing.

#### 14.5.4.6 Return Values

If the function succeeds, the return value is a pointer to the ContextList object.

#### 14.5.4.7 See Also

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::remove
```

## 14.5.5 CORBA::ContextList::remove

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

## 14.5.5.1 Synopsis

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

## 14.5.5.2 C++ Binding

Status remove(ULong index);

### 14.5.5.3 Argument

#### Index

The index into the ContextList object. The indexing is zero-based.

### 14.5.5.4 Exceptions

If this function does not succeed, the BAD PARAM exception is thrown.

# 14.5.5.5 Description

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

#### 14.5.5.6 Return Values

None.

#### 14.5.5.7 See Also

```
CORBA::ContextList::add
CORBA::ContextList::add_consume
CORBA::ContextList::count
CORBA::ContextList::item
```

# 14.6 NamedValue Member Functions

NamedValue is used only as an element of NVList, especially in the DII. NamedValue maintains an (optional) name, an any value, and labeling flags. Legal flag values are CORBA::ARG\_IN, CORBA::ARG\_OUT, and CORBA::ARG\_INOUT.

The value in a NamedValue may be manipulated via standard operations on any.

The mapping of these member functions to C++ is as follows:

```
// C++
class NamedValue
{
public:
Flags flags() const;
const char * name() const;
Any * value() const;
};
```

Memory Management

## 14.6.1 Memory Management

NamedValue has the following special memory management rule:

The index into the ContextList object. The indexing is zero-based.

• Ownership of the return values of the name() and value() functions is maintained by the NamedValue; these return values must not be freed by the caller.

The following sections describe NamedValue member functions.

- CORBA::NamedValue::flags
- CORBA::NamedValue::name
- CORBA::NamedValue::value

#### 14.6.1.1 CORBA::NamedValue::flags

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.6.1.1.1 Synopsis

Retrieves the flags attribute of the NamedValue object.

#### 14.6.1.1.2 C++ Binding

```
CORBA::Flags CORBA::NamedValue::flags () const;
```

#### 14.6.1.1.3 Arguments

None.

#### 14.6.1.1.4 Description

This member function retrieves the flags attribute of the NamedValue object.

#### 14.6.1.1.5 Return Values

If the function succeeds, the return value is the flags attribute of the NamedValue object.

If the function does not succeed, an exception is thrown.

#### 14.6.1.2 CORBA::NamedValue::name

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

#### 14.6.1.2.1 Synopsis

Retrieves the name attribute of the NamedValue object.

#### 14.6.1.2.2 C++ Binding

```
const char * CORBA::NamedValue::name () const;
```

#### 14.6.1.2.3 Arguments

None.

#### 14.6.1.2.4 Description

This member function retrieves the name attribute of the NamedValue object. The name returned by this member function is owned by the NamedValue object and must not be modified or released.

#### 14.6.1.2.5 Return Values

If the function succeeds, the value returned is a constant Identifier object representing the name attribute of the NamedValue object.

If the function does not succeed, an exception is thrown.

#### 14.6.1.3 CORBA::NamedValue::value

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

#### 14.6.1.3.1 Synopsis

Retrieves a pointer to the value attribute of the NamedValue object.

## 14.6.1.3.2 C++ Binding

```
CORBA::Any * CORBA::NamedValue::value () const;
```

#### 14.6.1.3.3 Arguments

None.

#### 14.6.1.3.4 Description

This member function retrieves a pointer to the Any object that represents the value attribute of the NamedValue object. This attribute is owned by the NamedValue object, and must not be modified or released.

#### 14.6.1.3.5 Return Values

If the function succeeds, the return value is a pointer to the Any object contained in the NamedValue object.

If the function does not succeed, an exception is thrown.

## 14.7 NVList Member Functions

NVList is a list of NamedValues. A new NVList is constructed using the <code>ORB::create\_list</code> operation (see <code>CORBA::ORB::create\_exception\_list</code>). New NamedValues may be constructed as part of an NVList, in any of following ways:

- add—creates an unnamed value, initializing only the flags
- add item—initializes name and flags

Each of these operations returns the new item.

Elements may be accessed and deleted via zero-based indexing. The add, add\_item, add\_value, add\_item\_consume, and add\_value\_consume functions lengthen the NVList to hold the new element each time they are called. The item function can be used to access existing elements.

```
// C++
class NVList
{
public:
ULong count() const;
NamedValue_ptr add(Flags);
NamedValue_ptr add_item(const char*, Flags);
NamedValue_ptr add_value(const char*, const Any&, Flags);
NamedValue_ptr item(ULong);
void remove(ULong);
};
```

Memory Management

## 14.7.1 Memory Management

NVList has the following special memory management rules:

- Ownership of the return values of the add, add\_item, add\_value, add\_item\_consume, add\_value\_consume, and item functions is maintained by the NVList; these return values must not be freed by the caller.
- The char\* parameters to the add\_item\_consume and add\_value\_consume functions and the Any\* parameter to the add\_value\_consume function are consumed by the NVList. The caller may not access these data after they have been passed to these functions because the NVList may copy them and destroy the originals immediately. The caller must use the NamedValue::value() operation to modify the value attribute of the underlying NamedValue, if desired.
- The remove function also calls CORBA::release on the removed NamedValue.

The following sections describe NVList member functions.

- CORBA::NVList::add
- CORBA::NVList::add\_item
- CORBA::NVList::add\_value
- CORBA::NVList::count

- CORBA::NVList::item
- CORBA::NVList::remove

#### 14.7.1.1 CORBA::NVList::add

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

#### 14.7.1.1.1 Synopsis

Constructs a NamedValue object with an unnamed item, setting only the flags attribute.

#### 14.7.1.1.2 C++ Binding

```
CORBA::NamedValue_ptr CORBA::NVList::add (
CORBA::Flags Flags);
```

#### 14.7.1.1.3 Argument

#### Flags

Flags to determine argument passing. Valid values are:

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

#### 14.7.1.1.4 Description

This member function constructs a NamedValue object with an unnamed item, setting only the flags attribute. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.

#### 14.7.1.1.5 Return Values

If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and must not be released.

If the member function does not succeed, a CORBA::NO MEMORY exception is thrown.

#### 14.7.1.1.6 See Also

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
```

CORBA::NVList::count
CORBA::NVList::remove

### 14.7.1.2 CORBA::NVList::add\_item

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

#### 14.7.1.2.1 Synopsis

Constructs a NamedValue object, creating an empty value attribute and initializing the name and flags attributes.

#### 14.7.1.2.2 C++ Binding

```
CORBA::NamedValue_ptr CORBA::NVList::add_item (
const char * Name,
CORBA::Flags Flags);
```

#### 14.7.1.2.3 Arguments

#### Name

The name of the list item.

#### Flags

Flags to determine argument passing. Valid values are:

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

#### 14.7.1.2.4 Description

This member function constructs a NamedValue object, creating an empty value attribute and initializing the name and flags attributes that pass in as parameters. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.

#### 14.7.1.2.5 Return Values

If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and must not be released.

If the member function does not succeed, an exception is thrown.

#### 14.7.1.2.6 See Also

```
CORBA::NVList::add

CORBA::NVList::add_value

CORBA::NVList::count

CORBA::NVList::item

CORBA::NVList::remove
```

## 14.7.1.3 CORBA::NVList::add value

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

#### 14.7.1.3.1 Synopsis

Constructs a NamedValue object, initializing the name, value, and flags attribute.

#### 14.7.1.3.2 C++ Binding

```
CORBA::NamedValue_ptr CORBA::NVList::add_value (
const char * Name,
const CORBA::Any & Value,
CORBA::Flags Flags);
```

#### 14.7.1.3.3 Arguments

#### Name

The name of the list item.

#### Value

The value of the list item.

#### Flags

Flags to determine argument passing. Valid values are:

```
CORBA::ARG_IN
CORBA::ARG_INOUT
CORBA::ARG_OUT
```

#### 14.7.1.3.4 Description

This member function constructs a NamedValue object, initializing the name, value, and flags attributes. The NamedValue object is added to the NVList object that the call was invoked upon.

The NVList object grows dynamically; your application does not need to track its size.

#### 14.7.1.3.5 Return Values

If the function succeeds, the return value is a pointer to the newly created NamedValue object. The returned NamedValue object reference is owned by the NVList and must not be released.

If the member function does not succeed, an exception is raised.

#### 14.7.1.3.6 See Also

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::count
CORBA::NVList::item
CORBA::NVList::remove
```

#### 14.7.1.4 CORBA::NVList::count

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

#### 14.7.1.4.1 Synopsis

Retrieves the current number of items in the list.

### 14.7.1.4.2 C++ Binding

```
CORBA::ULong CORBA::NVList::count () const;
```

#### 14.7.1.4.3 Arguments

None.

### 14.7.1.4.4 Description

This member function retrieves the current number of items in the list.

#### 14.7.1.4.5 Return Values

If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no NamedValue objects have been added, this function returns 0 (zero).

If the function does not succeed, an exception is thrown.

#### 14.7.1.4.6 See Also

CORBA::NVList::add CORBA::NVList::add\_item CORBA::NVList::add\_value CORBA::NVList::item CORBA::NVList::remove

#### 14.7.1.5 CORBA::NVList::item

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Values
- See Also

#### 14.7.1.5.1 Synopsis

Retrieves a pointer to the NamedValue object, based on the index passed in.

#### 14.7.1.5.2 C++ Binding

```
CORBA::NamedValue_ptr CORBA::NVList::item (
CORBA::ULong Index);
```

## 14.7.1.5.3 Argument

#### Index

The index into the NVList object. The indexing is zero-based.

### 14.7.1.5.4 Exception

If this function does not succeed, the BAD PARAM exception is thrown.

### 14.7.1.5.5 Description

This member function retrieves a pointer to a NamedValue object, based on the index passed in. The function uses zero-based indexing.

#### 14.7.1.5.6 Return Values

If the function succeeds, the return value is a pointer to the NamedValue object. The returned NamedValue object reference is owned by the NVList and must not be released.

#### 14.7.1.5.7 See Also

```
CORBA::NVList::add

CORBA::NVList::add_item

CORBA::NVList::add_value

CORBA::NVList::count

CORBA::NVList::remove
```

#### 14.7.1.6 CORBA::NVList::remove

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Values
- See Also

#### 14.7.1.6.1 Synopsis

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

## 14.7.1.6.2 C++ Binding

```
void CORBA::NVList::remove (
CORBA::ULong Index);
```

#### 14.7.1.6.3 Argument

#### Index

The index into the NVList object. The indexing is zero-based.

## 14.7.1.6.4 Exception

If this function does not succeed, the BAD PARAM exception is thrown.

### 14.7.1.6.5 Description

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

#### 14.7.1.6.6 Return Values

None.

#### 14.7.1.6.7 See Also

```
CORBA::NVList::add

CORBA::NVList::add_item

CORBA::NVList::add_value

CORBA::NVList::count

CORBA::NVList::item
```

# 14.8 Object Member Functions

The rules in this section apply to the OMG IDL interface Object, which is the base of the OMG IDL interface hierarchy. Interface Object defines a normal CORBA object, not a pseudo-object. However, it is included here because it references other pseudo-objects.

In addition to other rules, all operation names in interface Object have leading underscores in the mapped C++ class. Also, the mapping for <code>create\_request</code> is divided into three forms, corresponding to the usage styles described in the section Request Member Functions. The <code>is\_nil</code> and <code>release</code> functions are provided in the CORBA namespace, as described in Object Member Functions.

The Oracle Tuxedo software uses object reference operations that are defined by CORBA Revision 2.2. These operations depend only on type <code>Object</code>, so they can be expressed as regular functions within the CORBA namespace.



Because the Oracle Tuxedo software uses the POA and not the BOA, the deprecated <code>get\_implementation()</code> member function is not visible; you will get a compile error if you attempt to reference it.

The mapping of these member functions to C++ is as follows:

```
class CORBA
class Object
public:
CORBA::Boolean is a(const char *)
CORBA::Boolean is equivalent();
CORBA::Boolean nonexistent(Object ptr);
static Object ptr duplicate(Object ptr obj);
static Object ptr nil();
InterfaceDef ptr get interface();
CORBA::ULong hass(CORBA::ULong);
void create request (
Context ptr ctx,
const char *operation,
NVList ptr arg list,
NamedValue ptr result,
Request out request,
Flags req flags
```

```
Status _create_request(
Context_ptr ctx,
const char * operation,
NVList_ptr arg_list,
NamedValue_ptr result,
ExceptionList_ptr Except_list,
ContextList_ptr Context_list,
Request_out request,
Flags req_flags
);
Request_ptr _request(const char* operation);
}; //Object
}; // CORBA
```

The following sections describe the Object member functions.

- CORBA::Object::\_create\_request
- CORBA::Object::\_duplicate
- CORBA::Object::\_get\_interface
- CORBA::Object::\_is\_a
- CORBA::Object::\_is\_equivalent
- CORBA::Object::\_nil
- CORBA::Object::\_non\_existent
- CORBA::Object::\_request

## 14.8.1 CORBA::Object:: create request

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

### 14.8.1.1 Synopsis

Creates a request with user-specified information.

## 14.8.1.2 C++ Binding

```
Void CORBA::Object::_create_request (
CORBA::Context_ptr Ctx,
const char * Operation,
CORBA::NVList_ptr Arg_list,
CORBA::NamedValue_ptr Result,
CORBA::ExceptionList_ptr Except_list,
CORBA::ContextList_ptr Context_list,
```

```
CORBA::Request_out Request,
CORBA::Flags Req flags,);
```

## 14.8.1.3 Arguments

#### Ctx

The Context to be used for this request.

#### Operation

The operation name for this request.

#### Arg list

The argument list for this request.

#### Result

The NamedValue reference where the return value of this request is to be stored after a successful invocation.

#### Except list

The exception list for this request.

#### Context list

The context list for this request.

#### Request

The newly created request reference.

#### Req flags

Reserved for future use; the user must pass a value of zero.

## 14.8.1.4 Description

This member function creates a request that provides information on context, operation name, and other values (long form). To create a request with just the operation name supplied at the time of the call (short form), use the <code>CORBA::Object::\_request</code> member function. The remainder of the information provided in the long form eventually needs to be supplied.

### 14.8.1.5 Return Values

None.

#### 14.8.1.6 See Also

CORBA::Object:: request

# 14.8.2 CORBA::Object::\_duplicate

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values



#### Example

### 14.8.2.1 Synopsis

Duplicates the Object object reference.

## 14.8.2.2 C++ Binding

```
CORBA::Object_ptr CORBA::Object::_duplicate(
Object ptr Obj);
```

## 14.8.2.3 Argument

obj

The object reference to be duplicated.

### 14.8.2.4 Description

This member function duplicates the specified Object object reference (Obj.). If the given object reference is nil, the \_duplicate function returns a nil object reference. The object returned by this call must be freed using CORBA::release, or must be assigned to CORBA::Object\_var for automatic destruction.

This function can throw CORBA system exceptions.

#### 14.8.2.5 Return Values

Returns the duplicate object reference. If the specified object reference is nil, a nil object reference is returned.

## 14.8.2.6 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0","MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
```

# 14.8.3 CORBA::Object::\_get\_interface

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

## 14.8.3.1 Synopsis

Returns an interface definition for the Repository object.

### 14.8.3.2 C++ Binding

CORBA::InterfaceDef ptr CORBA::Object:: get interface ();

## 14.8.3.3 Arguments

None.

#### 14.8.3.4 Description

Returns an interface definition for the Repository object.



To use the Repository Interface API, define a macro before CORBA.h is included. For information about how to define a macro, see *Creating CORBA Server Applications*.

#### 14.8.3.5 Return Values

InterfaceDef\_ptr

# 14.8.4 CORBA::Object::\_is\_a

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example

## 14.8.4.1 Synopsis

Determines whether an object is of a certain interface.

## 14.8.4.2 C++ Binding

```
CORBA::Boolean CORBA::Object::_is_a(const char * interface_id);
```

# 14.8.4.3 Argument

#### interface id

A string that denotes the interface repository ID.

### 14.8.4.4 Description

This member function is used to determine if an object is an instance of the interface that you specify in the <code>interface\_id</code> parameter. It facilitates maintaining type-safety for object references over the scope of an ORB.

#### 14.8.4.5 Return Values

Returns TRUE if the object is an instance of the specified type, or if the object is an ancestor of the "most derived" type of that object.

### 14.8.4.6 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = op-> is a("IDL:Teller:1.0");
```

# 14.8.5 CORBA::Object::\_is\_equivalent

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- Example

## 14.8.5.1 Synopsis

Determines if two object references are equivalent.

## 14.8.5.2 C++ Binding

```
CORBA::Boolean CORBA::Object::_is_equivalent (
CORBA::Object ptr other obj);
```

### 14.8.5.3 Argument

#### other obj

The object reference for the other object, which is used for comparison with the target object.

## 14.8.5.4 Exceptions

Can throw a standard CORBA exception.

### 14.8.5.5 Description

This member function is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns  $\mathtt{TRUE}$  if your object reference is equivalent to the object reference you pass as a parameter. If two object references are identical, they are equivalent. Two different object references that refer to the same object are also equivalent.

#### 14.8.5.6 Return Values

Returns TRUE if the target object reference is known to be equivalent to the other object reference passed as a parameter; otherwise, it returns FALSE.

#### 14.8.5.7 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0", "MyTeller");
CORBA::Object_ptr dop = CORBA::Object::_duplicate(op);
CORBA::Boolean b = op-> is equivalent(dop);
```

# 14.8.6 CORBA::Object::\_nil

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- Example

# 14.8.6.1 Synopsis

Returns a reference to a nil object.

### 14.8.6.2 C++ Binding

```
CORBA::Object_ptr CORBA::Object::_nil();
```

## 14.8.6.3 Arguments

None.

## 14.8.6.4 Description

This member function returns a nil object reference. To test whether a given object is nil, use the appropriate  $CORBA::is\_nil$  member function (see the section CORBA::release). Calling the  $CORBA::is\_nil$  routine on any  $\_nil$  member function always yields  $CORBA\_TRUE$ .

#### 14.8.6.5 Return Values

Returns a nil object reference.

### 14.8.6.6 Example

```
CORBA::Object ptr op = CORBA::Object:: nil();
```

# 14.8.7 CORBA::Object::\_non\_existent

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.8.7.1 Synopsis

May be used to determine if an object has been destroyed.

## 14.8.7.2 C++ Binding

```
CORBA::Boolean CORBA::Object::_non_existent();
```

## 14.8.7.3 Arguments

None.

## 14.8.7.4 Description

This member function may be used to determine if an object has been destroyed. It does this without invoking any application-level operation on the object, and so will never affect the object itself.

#### 14.8.7.5 Return Values

Returns CORBA\_TRUE (rather than raising CORBA::OBJECT\_NOT\_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns CORBA\_FALSE.

## 14.8.8 CORBA::Object:: request

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values



#### See Also

## 14.8.8.1 Synopsis

Creates a request specifying the operation name.

## 14.8.8.2 C++ Binding

```
CORBA::Request_ptr CORBA::Object::_request (
const char * Operation);
```

## 14.8.8.3 Argument

#### Operation

The name of the operation for this request.

### 14.8.8.4 Description

This member function creates a request specifying the operation name. All other information, such as arguments and results, must be populated using CORBA::Request member functions.

#### 14.8.8.5 Return Values

If the member function succeeds, the return value is a pointer to the newly created request.

If the member function does not succeed, an exception is thrown.

#### 14.8.8.6 See Also

```
CORBA::Object::_create_request
```

## 14.9 CORBA Member Functions

This section describes the Object and Pseudo-Object Reference member functions.

The mapping of these member functions to C++ is as follows:

```
class CORBA {
void release(Object_ptr);
void release(Environment_ptr);
void release(NamedValue_ptr);
void release(NVList_ptr);
void release(Request_ptr);
void release(Context_ptr);
void release(TypeCode_ptr);
void release(POA_ptr);
void release(ORB_ptr);
void release(ExceptionList_ptr);
void release(ContextList_ptr);
boolean is_nil(Object_ptr);
Boolean is_nil(Environment_ptr);
```

```
Boolean is_nil(NamedValue_ptr);
Boolean is_nil(NVList_ptr);
Boolean is_nil(Request_ptr);
Boolean is_nil(Context_ptr);
Boolean is_nil(TypeCode_ptr);
Boolean is_nil(POA_ptr);
Boolean is_nil(ORB_ptr);
Boolean is_nil(ExceptionList_ptr);
Boolean is_nil(ContextList_ptr);
hash(maximum);
resolve_initial_references(identifier);
...
};
```

- CORBA::release
- CORBA::is nil
- CORBA::hash
- CORBA::resolve initial references

#### 14.9.1 CORBA::release

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example

## 14.9.1.1 Synopsis

Allows allocated resources to be released for the specified object type.

### 14.9.1.2 C++ Binding

```
void CORBA::release(spec_object_type obj);
```

## 14.9.1.3 Argument

ob j

The object reference that the caller will no longer access. The specified object type must be one of the types listed in the section .

## 14.9.1.4 Description

This member function indicates that the caller will no longer access the reference so that associated resources may be deallocated. If the specified object reference is nil, the release operation does nothing. If the ORB instance release is the last reference to the ORB, then the

ORB will be shut down prior to its destruction. This is the same as calling <code>ORB\_shutdown</code> prior to calling <code>CORBA::release</code>. This only applies to the <code>release</code> member function called on the ORB.

This member function may not throw CORBA exceptions.

#### 14.9.1.5 Return Values

None.

#### 14.9.1.6 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0", "MyTeller");
CORBA::release(op);
```

# 14.9.2 CORBA::is\_nil

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example

## 14.9.2.1 Synopsis

Determines if an object exists for the specified object type.

### 14.9.2.2 C++ Binding

```
CORBA::Boolean CORBA::is nil(spec object type obj);
```

## 14.9.2.3 Argument

obj

The object reference. The specified object type must be one of the types listed in the section CORBA Member Functions.

# 14.9.2.4 Description

This member function is used to determine if a specified object reference is nil. It returns TRUE if the object reference contains the special value for a nil object reference as defined by the ORB.

This operation may not throw CORBA exceptions.

#### 14.9.2.5 Return Values

Returns TRUE if the specified object is nil; otherwise, returns FALSE.

### 14.9.2.6 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0", "MyTeller");
CORBA::Boolean b = CORBA::is nil(op);
```

#### 14.9.3 CORBA::hash

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

## 14.9.3.1 Synopsis

Provides indirect access to object references using identifiers internal to the ORB.

### 14.9.3.2 C++ Binding

```
CORBA::hash(CORBA::ULong maximum);
```

## 14.9.3.3 Argument

#### maximum

Specifies an upper bound on the hash value returned by the ORB.

## 14.9.3.4 Description

Object references are associated with ORB-internal identifiers that may indirectly be accessed by applications using the hash() operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The maximum parameter to the hash operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision-chained hash table of object references, the more randomly distributed the values are within that range, and the less expensive those values are to compute, the better.

#### 14.9.3.5 Return Values

None.

# 14.9.4 CORBA::resolve\_initial\_references

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Values
- Example

### 14.9.4.1 Synopsis

Returns an initial object reference corresponding to an identifier string.

### 14.9.4.2 C++ Binding

```
CORBA::Object_ptr CORBA::resolve_initial_references(
const CORBA::char *identifier);
```

## 14.9.4.3 Argument

#### identifier

String identifying the object whose reference is required.

## 14.9.4.4 Exception

InvalidName

## 14.9.4.5 Description

Returns an initial object reference corresponding to an identifier string. Valid identifiers are "ROOTPOA" and "POACurrent".



This function is supported only for a joint client/server.

#### 14.9.4.6 Return Values

Returns a CORBA:: Object ptr.

## 14.9.4.7 Example

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
```

```
orb->resolve_initial_references("RootPOA");
PortableServer::POA_ptr rootPOA;
rootPOA = PortableServer::POA::narrow(pfobj);
```

## 14.10 ORB Member Functions

The ORB member functions constitute the programming interface to the Object Request Broker.

The mapping of the ORB member functions to C++ is as follows:

```
class CORBA
class ORB
public:
char *object to string(Object ptr);
Object ptr string to object(const char *);
void create list(Long, NVList out);
void create operation list (operationDef ptr, NVList out);
void create named value(NamedValue out);
void create exception list(ExceptionList out);
void create context list(ContextList out);
void get default context (Context out);
void create environment(Environment out);
void send multiple requests oneway(const requestSeq&);
void send multiple requests deferred(const requestSeq&);
Boolean poll next response();
void get next response (Request out);
Boolean work pending();
void perform work();
void create policy (in PolicyType type, in any val);
// Extension
void destroy();
// Extensions to support sharing context between threads
void Ctx get ctx() = 0;
void set ctx(Ctx) = 0;
void clear ctx() = 0;
// Thread extensions
void inform thread exit(TID) = 0;
}; //ORB
}; // CORBA
```

#### **Thread-related Operations:**

To support single-threaded ORBs, as well as multithreaded ORBs that run multithread-unaware code, two operations (perform\_work and work\_pending) are included in the ORB interface. These operations can be used by single-threaded and multithreaded applications. An application that is a pure ORB client would not need to use these operations.

To support multithreaded server applications, four operations (get\_ctx, set\_ctx, clear\_ctx, and inform thread exit) are included as extensions to the ORB interface.

The following sections describe the ORB member functions.

CORBA::ORB::clear ctx



- CORBA::ORB::create context list
- CORBA::ORB::create\_environment
- CORBA::ORB::create\_exception\_list
- CORBA::ORB::create\_list
- CORBA::ORB::create named value
- CORBA::ORB::create\_operation\_list
- CORBA::ORB::create\_policy
- CORBA::ORB::destroy
- CORBA::ORB::get\_ctx
- CORBA::ORB::get\_default\_context
- CORBA::ORB::get\_next\_response
- CORBA::ORB::inform thread exit
- CORBA::ORB::list\_initial\_services
- CORBA::ORB::object\_to\_string
- CORBA::ORB::perform work
- CORBA::ORB::poll\_next\_response
- CORBA::ORB::resolve initial references
- CORBA::ORB::send\_multiple\_requests\_deferred
- CORBA::ORB::send multiple requests oneway
- CORBA::ORB::set\_ctx
- CORBA::ORB::string\_to\_object
- CORBA::ORB::work\_pending

# 14.10.1 CORBA::ORB::clear\_ctx

- Synopsis
- C++ Binding
- Parameters
- Return Value
- Description
- Example
- See Also

## 14.10.1.1 Synopsis

Indicates that a context is no longer required by this thread. This method supports the development of a multithreaded server application.

## 14.10.1.2 C++ Binding

void clear ctx()

#### 14.10.1.3 Parameters

None.

#### 14.10.1.4 Return Value

None.

### 14.10.1.5 Description

This method is called by an application-managed thread after the thread has finished using the context. The method removes the association between that thread and a context.



Do not call the <code>clear\_ctx</code> method from within a thread that is managed by the Oracle Tuxedo system. The Oracle Tuxedo system performs the appropriate context propagation and cleanup automatically for the threads it manages. If this method is called on a thread managed by the Oracle Tuxedo system, the <code>BAD\_PARAM</code> exception is thrown.

#### 14.10.1.6 Example

```
TP::orb()->clear_ctx();
```

## 14.10.1.7 See Also

```
CORBA::ORB::get_ctx
CORBA::ORB::set_ctx
```

# 14.10.2 CORBA::ORB::create\_context\_list

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

## 14.10.2.1 Synopsis

Creates and returns a list of contexts.

### 14.10.2.2 C++ Binding

```
void CORBA::ORB::create_context_list(
CORBA::ContextList_out List);
```

### 14.10.2.3 Argument

List

Receives a reference to the newly created context list.

#### 14.10.2.4 Description

This member function creates and returns a list of context strings that must be supplied with the Request operation in a form that may be used in the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the CORBA::release member function.

#### 14.10.2.5 Return Values

None.

## 14.10.3 CORBA::ORB::create\_environment

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

### 14.10.3.1 Synopsis

Creates an environment.

## 14.10.3.2 C++ Binding

```
void CORBA::ORB::create_environment (
CORBA::Environment out New env);
```

## 14.10.3.3 Argument

New env

Receives a reference to the newly created environment.

## 14.10.3.4 Description

This member function creates an environment.

#### 14.10.3.5 Return Values

None.

#### 14.10.3.6 See Also

CORBA::NVList::add
CORBA::NVList::add\_item
CORBA::NVList::add\_value
CORBA::release

# 14.10.4 CORBA::ORB::create\_exception\_list

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

#### 14.10.4.1 Synopsis

Returns a list of exceptions.

### 14.10.4.2 C++ Binding

```
void CORBA::ORB::create_exception_list(
CORBA::ExceptionList out List);
```

# 14.10.4.3 Argument

List

Receives a reference to the newly created exception list.

# 14.10.4.4 Description

This member function creates and returns a list of exceptions in a form that may be used in the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the CORBA::release member function.

## 14.10.4.5 Return Values

None.

# 14.10.5 CORBA::ORB::create\_list

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

#### See Also

## 14.10.5.1 Synopsis

Creates and returns an NVList object reference.

## 14.10.5.2 C++ Binding

```
void CORBA::ORB::create_list (
CORBA::Long NumItem,
CORBA::NVList out List);
```

### 14.10.5.3 Arguments

#### NumItem

The number of elements to preallocate in the newly created list.

#### List

Receives the newly created list.

### 14.10.5.4 Description

This member function creates a list, preallocating a specified number of items. List items may be sequentially added to the list using the <code>CORBA::NVList\_add\_item</code> member function. When no longer needed, this list must be freed using the <code>CORBA::release</code> member function.

#### 14.10.5.5 Return Values

None.

#### 14.10.5.6 See Also

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::release
```

## 14.10.6 CORBA::ORB::create\_named\_value

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

# 14.10.6.1 Synopsis

Creates a NamedValue object reference.

### 14.10.6.2 C++ Binding

```
void CORBA::ORB::create_named_value (
NameValue out NewNamedVal);
```

### 14.10.6.3 Argument

#### NewNamedVal

A reference to the newly created NamedValue object.

## 14.10.6.4 Description

This member function creates a NamedValue object. Its intended use is for the result argument of a request that needs a NamedValue object. The extra steps of creating an NVList object are avoided by calling this member function.

When no longer needed, the NamedValue object must be freed using the CORBA::release member function.

#### 14.10.6.5 Return Values

None.

#### 14.10.6.6 See Also

```
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::release
```

# 14.10.7 CORBA::ORB::create\_operation\_list

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

## 14.10.7.1 Synopsis

Creates and returns a list of the arguments of a specified operation.

### 14.10.7.2 C++ Binding

```
void CORBA::ORB::create_operation_list (
CORBA::OperationDef_ptr Oper,
CORBA::NVList out List);
```

#### 14.10.7.3 Arguments

#### Oper

The operation definition for which the list is being created.

#### List

Receives a reference to the newly created arguments list.

## 14.10.7.4 Description

This member function creates and returns a list of the arguments of a specified operation, in a form that may be used with the Dynamic Invocation Interface (DII). When no longer needed, this list must be freed using the CORBA::release member function.

### 14.10.7.5 Return Values

None.

#### 14.10.7.6 See Also

```
CORBA::OBB::create_list
CORBA::NVList::add
CORBA::NVList::add_item
CORBA::NVList::add_value
CORBA::release
```

# 14.10.8 CORBA::ORB::create\_policy

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- Example

## 14.10.8.1 Synopsis

Creates new instances of policy objects of a specific type with specified initial state.

### 14.10.8.2 C++ Binding

```
void CORBA::ORB::create_policy (
in PolicyType type,
in any val);
```

#### 14.10.8.3 Arguments

#### type

BiDirPolicy::BIDIRECTIONAL\_POLICY\_TYPE is the only PolicyType value supported for Oracle WebLogic Enterprise version 4.2.

#### val

The only val value supported for Oracle WebLogic Enterprise V4.2 is BiDirPolicy::BidirectionalPolicyValue.

### 14.10.8.4 Exceptions

#### PolicyError

This exception is raised to indicate problems with the parameter values passed to the <code>ORB::create\_policy</code> operation. The specific exception and reasons are as follows shown in the following table:

| Exception                | Reason   |
|--------------------------|--|
| BAD_POLICY               | The requested Policy is not understood by the ORB.   |
| UNSUPPORTED_POLICY       | The requested Policy is understood to be valid by the ORB, but is not currently supported.   |
| BAD_POLICY_TYPE          | The type of the value requested for the Policy is not valid for that PolicyType.   |
| BAD_POLICY_VALUE         | The value requested for the Policy is of a valid type, but is not within the valid range for that type.  |
| UNSUPPORTED_POLICY_VALUE | The value requested for the Policy is of a valid type and within the valid range for that type, but this valid value is not currently supported. |

# 14.10.8.5 Description

This operation can be invoked to create new instances of policy objects of a specific type with specified initial state. If <code>create\_policy</code> fails to instantiate a new Policy object due to its inability to interpret the requested type and content of the policy, it raises the Policy Error exception with the appropriate reason. (See Exceptions below.)

The BidirectionalPolicy argument is provided for remote clients using callbacks because remote clients use IIOP. It is not used for native clients using callbacks or for Oracle Tuxedo servers because machines inside an Oracle Tuxedo domain communicate differently.

Before GIOP 1.2, bidirectional policy was not available as a choice in IIOP (which uses TCP/IP). Connections in GIOP 1.0 and 1.1 were one way (that is, a request flowed from a client to a

server); only responses flowed from the server back to the client. If the server wanted to make a request back to the client machine (say for a callback), the server machine had to establish another one-way connection. (Be advised that "connections" in this sense mean operating system resources, not physically different wires or communication paths. A connection uses resources, so minimizing connections is desirable.)

Since this release of the Oracle Tuxedo C++ software supports GIOP 1.2, it supports reuse of the TCP/IP connection for both incoming and outgoing requests. Reusing connections saves resources when a remote client sends callback references to an Oracle Tuxedo domain. The joint client/server uses a connection to send a request to an Oracle Tuxedo domain; that connection can be reused for the callback request. If the connection is not reused, the callback request must establish another connection.

Allowing reuse of a connection is a choice of the ORB/POA that creates callback object references. The server for those object references (usually the creator of the references, especially in the callback case) might choose not to allow reuse for security considerations (that is, the outgoing connection [a client request from this machine to a remote server] may not need security because the remote server does not require it, but the callback server on this machine might require security). Since security is established partly on a connection basis, the incoming security can be established only if a separate connection is used. If the remote server requires security, and if that security involves a mutual authentication, the local server usually feels safe in allowing reuse of the connection.

Since the choice of connection reuse is at the server end, whenever a process acts as a server—in this case a joint client/server—and creates object references, it must inform the ORB that it is willing to reuse connections. The process does this by setting a policy on the POA that creates the object references. The default policy is to not allow reuse (that is, if you do not supply a policy object for reuse, the POA does not allow reuse).

This default allows for backward compatibility with code written before CORBA version 2.3. Such code did not know that reuse was possible so it did not have to take into consideration the security implications of reuse. Thus, that unchanged code must continue to disallow reuse until the user considers security and explicitly makes a decision to the contrary.

To allow reuse, you use the <code>create\_policy</code> operation to create a policy object that allows reuse, and use that policy object as part of the list of policies for POA creation.

#### 14.10.8.6 Return Values

None.

## 14.10.8.7 Example

```
#include <BiDirPolicy_c.h>
BiDirPolicy::BidirectionalPolicy_var bd_policy;
CORBA::Any allow_reuse;
allow_reuse <<= BiDirPolicy::BOTH;
CORBA::Policy_var generic_policy =
orb->create_policy( BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
allow_reuse );
bd_policy = BiDirPolicy::BidirectionalPolicy::_narrow(
generic policy);
```

In the above example, the  $bd\_policy$  would then be placed in the PolicyList passed to the create poa operation.

## 14.10.9 CORBA::ORB::destroy

- Synopsis
- C++ Binding
- Parameter
- Return Value
- Description
- Example

## 14.10.9.1 Synopsis

Destroys the specified ORB.

## 14.10.9.2 C++ Binding

void destroy();

#### 14.10.9.3 Parameter

None.

#### 14.10.9.4 Return Value

None.

### 14.10.9.5 Description

Use this method to destroy an ORB so that the resources associated with that ORB can be reclaimed. Once an ORB has been destroyed, another invocation on the <code>ORB\_init</code> method with the same ORB ID returns a reference to a newly constructed ORB. If an application invokes the <code>ORB::destroy</code> method from a thread that is currently servicing an invocation, the Oracle Tuxedo system raises the <code>BAD\_INV\_ORDER</code> system exception with the OMG minor code 3, because blocking would result in a deadlock.

## 14.10.9.6 Example

pOrb->destroy();

# 14.10.10 CORBA::ORB::get\_ctx

- Synopsis
- C++ Binding
- Arguments
- Return Value
- Description



- Example
- See Also

## 14.10.10.1 Synopsis

Retrieves the context associated with the current thread. This method supports the development of a multithreaded server application.

## 14.10.10.2 C++ Binding

```
CORBA::ORB::Ctx get ctx()
```

### 14.10.10.3 Arguments

None.

#### 14.10.10.4 Return Value

```
CORBA::ORB::Ctx
```

The context associated with this thread.

### 14.10.10.5 Description

Use this method to retrieve the context associated with the current thread. This context can then be used to initialize other threads that the application creates and manages.

When an object creates a thread, the object invokes this operation on the ORB to obtain system context information that the object can pass on to the thread. This operation must be called from a thread that already has a context. For example, the thread in which a method was dispatched will already have a content.

## 14.10.10.6 Example

```
thread.context = TP::orb()->get ctx();
```

#### 14.10.10.7 See Also

```
CORBA::ORB::set_ctx
CORBA::ORB::clear ctx
```

# 14.10.11 CORBA::ORB::get\_default\_context

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

### 14.10.11.1 Synopsis

Returns a reference to the default context.

## 14.10.11.2 C++ Binding

```
void CORBA::ORB::get_default_context (
CORBA::Context out ContextObj);
```

## 14.10.11.3 Argument

#### ContextObj

The reference to the default context.

## 14.10.11.4 Description

This member function returns a reference to the default context. When no longer needed, this context reference must be freed using the CORBA::release member function.

### 14.10.11.5 Return Values

None.

# 14.10.11.6 See Also

```
CORBA::Context::get_one_value
CORBA::Context::get values
```

# 14.10.12 CORBA::ORB::get\_next\_response

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

# 14.10.12.1 Synopsis

Determines and reports the next deferred synchronous request that completes.

# 14.10.12.2 C++ Binding

```
void CORBA::ORB::get_next_response (
CORBA::Request out RequestObj);
```

### 14.10.12.3 Argument

#### RequestObj

The reference to the next completed request.

### 14.10.12.4 Description

This member function returns a reference to the next request that completes. If no requests have completed, the function waits for a request to complete. This member function returns the next request on the queue, in contrast to the <code>CORBA::Request::get\_response</code> member function, which waits for a particular request to complete. When no longer needed, this request must be freed using the <code>CORBA::release</code> member function.

### 14.10.12.5 Return Values

None.

### 14.10.12.6 See Also

```
CORBA::ORB::poll_next_response
CORBA::Request::get reponse
```

## 14.10.13 CORBA::ORB::inform thread exit

- Synopsis
- C++ Binding
- Parameter
- Return Value
- Description
- Example

## 14.10.13.1 Synopsis

Informs the Oracle Tuxedo system that resources associated with an application-managed thread can be released. This method supports the development of a multithreaded server application.

# 14.10.13.2 C++ Binding

```
void CORBA::ORB::inform thread exit(CORBA::TID threadId)
```

### 14.10.13.3 Parameter

#### threadId

The logical thread ID of the application-managed thread being deleted.

### 14.10.13.4 Return Value

None.

## 14.10.13.5 Description

This method informs the Oracle Tuxedo system about the following conditions:

- The specified application-managed thread is no longer used by a servant implementation.
- Any resources associated with the thread must be released.



You must only call this operation on threads that the application creates and manages. Do not invoke this method when specifying a dispatch thread that is managed by the Oracle Tuxedo system.

### 14.10.13.6 Example

pOrb->inform\_thread\_exit(thread.threadId);

# 14.10.14 CORBA::ORB::list\_initial\_services

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

### 14.10.14.1 Synopsis

Determines which objects have references available via the initial references mechanism.

## 14.10.14.2 C++ Binding

```
typedef string ObjectId;
typedef sequence ObjectId ObjectIdList;
ObjectIdList list initial services ();
```

# 14.10.14.3 Argument

ObjectId
The object ID.

list\_initial\_services ()
Defines the object type.

## 14.10.14.4 Description

This operation is used by applications to determine which objects have references available via the initial references mechanism. This operation returns an <code>ObjectIdList</code>, which is a sequence of <code>ObjectIds</code>. <code>ObjectIds</code> are typed as strings.

Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the ID, the type of object being returned must be defined, that is, InterfaceRepository returns an object of type Repository, and NameService returns a CosNamingContext object.

### 14.10.14.5 Return Values

Sequence of ObjectIds.

### 14.10.14.6 See Also

```
CORBA::ORB::resolve_initial_references
```

# 14.10.15 CORBA::ORB::object to string

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example
- See Also

# 14.10.15.1 Synopsis

Produces a string representation of an object reference.

## 14.10.15.2 C++ Binding

```
char * CORBA::ORB::object_to_string (
CORBA::Object ptr ObjRef);
```

### 14.10.15.3 Argument

#### ObjRef

The object reference to represent as a string.

### 14.10.15.4 Description

This member function produces a string representation of an object reference. The calling program must use the <code>CORBA::string\_free</code> member function to free the string memory after it is no longer needed.

### 14.10.15.5 Return Values

The string representing the specified object reference.

### 14.10.15.6 Example

```
CORBA::Object_ptr op = TP::create_object_reference(
"IDL:Teller:1.0", "MyTeller");
char* objstr = TP::orb()->object to string(op);
```

### 14.10.15.7 See Also

```
CORBA::ORB::string_to_object
CORBA::string free
```

# 14.10.16 CORBA::ORB::perform\_work

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- See Also
- Example

## 14.10.16.1 Synopsis

Allows the ORB to perform server-related work.

## 14.10.16.2 C++ Binding

```
void CORBA::ORB::perform work ();
```

### 14.10.16.3 Arguments

None.

### 14.10.16.4 Exceptions

Once the ORB has shut down, a call to  $work\_pending$  and  $perform\_work$ () raises the BAD\_INV\_ORDER exception. An application can detect this exception to determine when to terminate a polling loop.

### 14.10.16.5 Description

If called by the main thread, this operation allows the ORB to perform server-related work. Otherwise, it does nothing.

The work\_pending() and perform\_work() operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multithreaded server would need a polling loop only if there were both ORB and other code that required use of the main thread. See the example below for such a polling loop.

### 14.10.16.6 Return Values

None.

### 14.10.16.7 See Also

```
CORBA::ORB::work pending
```

### 14.10.16.8 Example

The following is an example of a polling loop:

```
// C++
for (;;) {
if (orb->work_pending()) {
  orb->perform_work();
}
// do other things
// sleep?
}
```

# 14.10.17 CORBA::ORB::poll\_next\_response

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also



### 14.10.17.1 Synopsis

Determines whether a completed request is outstanding.

### 14.10.17.2 C++ Binding

```
CORBA::Boolean CORBA::ORB::poll next response ();
```

# 14.10.17.3 Arguments

None.

## 14.10.17.4 Description

This member function reports on whether there is an outstanding (pending) completed request; it does not remove the request. If a completed request is outstanding, the next call to the CORBA::ORB::get\_next\_response member function is guaranteed to return a request without waiting. If there are no completed requests outstanding, the CORBA::ORB::poll\_next\_response member function returns without waiting (blocking).

### 14.10.17.5 Return Values

If a completed request is outstanding, the function returns CORBA TRUE.

If no completed request is outstanding, the function returns CORBA FALSE.

### 14.10.17.6 See Also

```
CORBA::ORB::get next response
```

# 14.10.18 CORBA::ORB::resolve\_initial\_references

- Synopsis
- · C++ Binding
- Augument
- Description
- Return Values
- See Also

## 14.10.18.1 Synopsis

Obtains object references for initial services.

# 14.10.18.2 C++ Binding

```
Object resolve_initial_references ( in ObjectId identifier )
raises (InvalidName);
```

exception InvalidName {};

### 14.10.18.3 Augument

#### identifier

String that identifies the object whose reference is required.

## 14.10.18.4 Description

This operation is used by applications to obtain object references for initial services. The interface differs from the Naming Service's resolve in that <code>ObjectId</code> (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the namespace to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this means. Unlike the ORB identifiers, the ObjectId name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved ObjectIds are RootPOA, POACurrent, InterfaceRepository, NameService, TradingService, SecurityCurrent, TransactionCurrent, and DynAnyFactory.

The application is responsible for narrowing the object reference returned from resolve\_initial\_references to the type that was requested in the <code>ObjectId</code>. For example, for <code>InterfaceRepository</code> the object returned would be narrowed to <code>Repository</code> type.

### 14.10.18.5 Return Values

Object references for initial services.

### 14.10.18.6 See Also

```
CORBA::ORB::list initial services
```

# 14.10.19 CORBA::ORB::send\_multiple\_requests\_deferred

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

## 14.10.19.1 Synopsis

Sends a sequence of deferred synchronous requests.

### 14.10.19.2 C++ Binding

```
void CORBA::ORB::send_multiple_requests_deferred (
const CORBA::ORB::RequestSeq & Reqs);
```

### 14.10.19.3 Argument

#### Reqs

The sequence of requests to be sent. For more information about how to populate the sequence with request references, see CORBA::ORB::RequestSeq in the section Usage.

### 14.10.19.4 Description

This member function sends out a sequence of requests and returns control to the caller without waiting for the operation to complete. The caller uses

```
CORBA::ORB::poll_next_response, CORBA::ORB::get_next_response, Or CORBA::Rquest::get_response or all three to determine if the operation has completed and if the output arguments have been updated.
```

### 14.10.19.5 Return Values

None.

### 14.10.19.6 See Also

```
CORBA::Request::get_response
CORBA::ORB::get_next_response
CORBA::ORB::send multiple requests oneway
```

# 14.10.20 CORBA::ORB::send\_multiple\_requests\_oneway

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

## 14.10.20.1 Synopsis

Sends a sequence of one-way, deferred synchronous requests.

### 14.10.20.2 C++ Binding

```
void CORBA::ORB::send_multiple_requests_oneway (
const CORBA::RequestSeq & Reqs);
```

### 14.10.20.3 Argument

#### Regs

The sequence of requests to be sent. For more information about how to populate the sequence with request references, see <code>CORBA::ORB::RequestSeq</code> in the section Usage.

### 14.10.20.4 Description

This member function sends out a sequence of requests and returns control to the caller without waiting for the operation to complete. The caller neither intends to wait for a response nor expects any output arguments to be updated.

### 14.10.20.5 Return Values

None.

### 14.10.20.6 See Also

CORBA::ORB::send multiple requests deferred

# 14.10.21 CORBA::ORB::set\_ctx

- Synopsis
- C++ Binding
- Parameter
- Return Value
- Description
- Example
- See Also

# 14.10.21.1 Synopsis

Sets the context for the current thread. This method supports the development of a multithreaded server application.

# 14.10.21.2 C++ Binding

void set ctx(CORBA::ORB::Ctx aContext)

### 14.10.21.3 Parameter

#### aContext

The context to be associated with this thread.

### 14.10.21.4 Return Value

None.

### 14.10.21.5 Description

This method sets the context for the current application-managed thread. The context parameter provided must have been obtained in a previously-executed thread that is managed by the Oracle Tuxedo system or in an application-managed thread that has already been initialized.



Do not call the <code>set\_ctx</code> method in a thread that is managed by the Oracle Tuxedo system. The Oracle Tuxedo system performs the appropriate context propagation automatically for the threads it manages. If your application calls this method on a thread managed by the Oracle Tuxedo system, the <code>BAD PARAM</code> exception is thrown.

### 14.10.21.6 Example

```
TP::orb()->set ctx(thread->context);
```

### 14.10.21.7 See Also

```
CORBA::ORB::get_ctx()
CORBA::ORB::clear ctx()
```

# 14.10.22 CORBA::ORB::string\_to\_object

- Synopsis
- · C++ Binding
- Argument
- Description
- Return Value
- See Also

## 14.10.22.1 Synopsis

Converts a string produced by CORBA::ORB::object\_to\_string operation and returns the corresponding object reference.

## 14.10.22.2 C++ Binding

```
Object string_to_object ( in string str );
```

### 14.10.22.3 Argument

str

String produced by the CORBA::ORB::object to string operation.

### 14.10.22.4 Description

This operation is used by applications to convert a string produced by CORBA::ORB::object to string operation and returns the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB's object\_to\_string operation must be used to produce the string. The string\_to\_object operation allows URLs in the IOR, corbaloc, corbalocs, and corbanames formats to be converted into object references. If a conversion fails, the string\_to\_object operation raises the BAD PARAM standard exception with one of the following minor codes:

- BadSchemeName
- BadAddress
- BadSchemeSpecificPart

For all conforming ORBs, if <code>obj</code> is a valid reference to an object, then <code>string\_to\_object(object\_to\_string(obj))</code> will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

### 14.10.22.5 Return Value

Returns an object reference.

### 14.10.22.6 See Also

CORBA::ORB::object to string

# 14.10.23 CORBA::ORB::work\_pending

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

## 14.10.23.1 Synopsis

Returns an indication of whether the ORB needs the main thread to perform server-related work.



# 14.10.23.2 C++ Binding

```
CORBA::boolean CORBA::ORB::work_pending ();
```

### 14.10.23.3 Arguments

None.

### 14.10.23.4 Description

This operation returns an indication of whether the ORB needs the main thread to perform server-related work.

### 14.10.23.5 Return Values

A result of TRUE indicates that the ORB needs the main thread to perform server-related work, and a result of FALSE indicates that the ORB does not need the main thread.

### 14.10.23.6 See Also

```
CORBA::ORB::perform work
```

# 14.11 ORB Initialization Member Function

The mapping of this member function to C++ is as follows:

```
class CORBA {
  static CORBA::ORB_ptr ORB_init(int& argc, char** argv,
  const char* orb_identifier = 0,
  const char* -ORBport nnn);
  <appl-name> [-ORBid {BEA_IIOP | BEA_TOBJ} \
  [-ORBInitRef <ObjectID>=<ObjectURL> [*]]
  [-ORBport port-number] \
  [-ORBsecurePort port-number] \
  [-ORBminCrypto {0 | 40 | 56 | 128}] \
  [-ORBmaxCrypto {0 | 40 | 56 | 128}] \
  [-ORBmutualAuth] \
  [-ORBpeerValidate {detect | warn | none}] \
  [appl-options]
};
```

- CORBA::ORB\_init
- ORB

# 14.11.1 CORBA::ORB\_init

- Synopsis
- C++ Binding

- Arguments
- Description
- Return Value
- Exceptions

### 14.11.1.1 Synopsis

Initializes operations for an ORB.

### 14.11.1.2 C++ Binding

```
static CORBA::ORB_ptr ORB_init( int& argc, char** argv,
const char* orb identifier = 0);
```

### 14.11.1.3 Arguments

#### argc

The number of strings in argv.

#### arqv

This argument is defined as an unbound array of strings (char \*\*) and the number of strings in the array is passed in the argc parameter.

#### orb identifier

If the orb\_identifier parameter is supplied, "BEA\_IIOP" explicitly specifies a remote client and "BEA\_TOBJ" explicitly specifies a native client, as defined in the section Tobj Bootstrap.

## 14.11.1.4 Description

This member function initializes operations for an ORB and returns a pointer to the ORB. When your program is done with the ORB, use the <code>CORBA::release</code> member function to free the resources allocated for the ORB pointer returned from <code>CORBA::ORB\_ptr ORB\_init</code>.

The ORB returned has been initialized with two pieces of information to determine how it will operate: client type (remote or native) and server port number. The client type can be specified in the orb\_identifier argument, in the argv argument, or in the system registry. The server port number can be specified in the argv argument.

The arguments <code>argc</code> and <code>argv</code> are typically the same parameters that were passed to the main program. As specified by C++, these parameters contain string tokens from the command line that started the client. The two ORB options can be specified on the command line, each using a pair of tokens, as shown in examples below.

#### **Client Type**

The ORB init function determines the client type of the ORB by the following steps.

- 1. If the orb\_identifier argument is present, ORB\_init determines the client type, either native or remote, if the string is "BEA\_IIOP" or "BEA\_TOBJ", respectively. If an orb identifier string is present, all -ORBid parameters in the argy are ignored (removed).
- 2. If orb\_identifier is not present or is explicitly zero, ORB\_init looks at the entries in argc/argv. If argv contains an entry with "-ORBid", the next entry must be either "BEA IIOP" or

- "BEA\_TOBJ", again specifying remote or native. This pair of entries occurs if the command line contains either "-ORBid BEA IIOP" or "-ORBid BEA TOBJ".
- 3. If no client type is specified in argc/argv, ORB\_init uses the default client type from the system registry (BEA\_IIOP or BEA\_TOBJ). The system registry was initialized at the time Oracle Tuxedo was installed

#### **Server Port**

In the case of an Oracle Tuxedo remote joint client/server, in order to support IIOP, by definition, the object references created for the server part must contain a host and port. For transient object references, any port is sufficient and can be obtained by the ORB dynamically, but this is not sufficient for persistent object references. Persistent references must be served on the same port after the ORB restarts, that is, the ORB must be prepared to accept requests on the same port with which it created the object reference. Thus, there must be some way to configure the ORB to use a particular port.

Typically, a system administrator assigns the port number for the client from the "user" range of port numbers rather from the dynamic range. This keeps the joint client/servers from using conflicting ports.

To determine port number, <code>ORB\_init</code> searches the <code>argv</code> parameter for the token "<code>-ORBport</code>" and a following numeric token. For example, if the client executable is named sherry, the command line might specify that the server port must be 937 as follows:

sherry -ORBport 937

#### **ARGV Parameter Considerations**

For C++, the order of consumption of argv parameters may be significant to an application. To ensure that applications are not required to handle argv parameters they do not recognize, the ORB initialization function must be called before the remainder of the parameters are consumed. Therefore, after the ORB\_init call, the argv and argc parameters have been modified to remove the ORB understood arguments. It is important to note that the ORB\_init function can only reorder or remove references to parameters from the argv list. This restriction is made to avoid potential memory management problems caused by trying to free parts of the argv list or extending the argv list of parameters. This is why argv is passed as a char\*\* and not as a char\*\* \$\parameters\$.



Use the CORBA::release member function to free the resources allocated for the pointer returned from CORBA::ORB init.

### 14.11.1.5 Return Value

A pointer to a CORBA::ORB.

### 14.11.1.6 Exceptions

None.

### 14.11.2 ORB

- Synopsis
- Syntax
- Description
- Parameters
- Portability
- Interoperability
- Examples
- See Also

# 14.11.2.1 Synopsis

Configures applications based on the Oracle Tuxedo CORBA C++ ORB to access or provide Oracle Tuxedo CORBA objects.

### 14.11.2.2 Syntax

```
<appl-name> [-ORBid {BEA_IIOP | BEA_TOBJ} \
[-ORBInitRef <ObjectID>=<ObjectURL> [*]]
[-ORBDefaultInitRef <ObjectURL>]
[-ORBport port-number] \
[-ORBsecurePort port-number] \
[-ORBminCrypto {0 | 40 | 56 | 128}] \
[-ORBmaxCrypto {0 | 40 | 56 | 128}] \
[-ORBmutualAuth] \
[-ORBpeerValidate {detect | warn | none}] \
[appl-options]
```

## 14.11.2.3 Description

The Oracle Tuxedo CORBA C++ ORB is an Oracle Tuxedo-supplied library that enables the development of CORBA-based applications used to access or provide Oracle Tuxedo objects using IIOP or IIOP-SSL. The ORB command-line options allow for customization.

### 14.11.2.4 Parameters

```
[-ORBid {BEA IIOP | BEA TOBJ}]
```

The value BEA\_IIOP explicitly specifies that the ORB be configured to support either a client or a server environment that communicates over the IIOP or IIOP-SSL protocol.

The value BEA\_TOBJ explicitly specifies that the ORB be configured to support the native client environment that can only communicate over the TGIOP protocol within an Oracle Tuxedo domain.

If not specified, the ORB will detect the environment in which it is deployed and configure itself for use in that environment.

#### [-ORBInitRef ObjectId=ObjectURL]

The ORB initial reference argument, -ORBInitRef, allows specification of an arbitrary object reference for an initial service.

ObjectID represents the well-known object ID for a service that is defined in the CORBA specification. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed.

Objecturl can be any of the URL schemes supported by the <code>CORBA::ORB::string\_to\_object</code> operation as defined in CORBA specification. If a URL is syntactically malformed or can be determined to be invalid in an implementation-defined manner, <code>CORBA::ORB\_init</code> will raise the <code>CORBA::BAD\_PARAM</code> standard exception listed in the following table.

| Minor Code                | Description  |
|---------------------------|--|
| BadSchemeSpe<br>cificPart | The specified scheme is recognized by the ORB implementation. Only the schemes IOR, corbaloc, corbalocs, and corbaname are supported.                                  |
| BadAddress                | The format of the address is not recognized by the ORB implementation. Host names must be specified according to DNS or as class C IP addresses in dot-separated form. |
| BadSchemeSpe<br>cificPart | The format of the address is not recognized by the ORB implementation. Host names must be specified according to DNS or as class C IP addresses in dot-separated form. |
| BadSchemeSpe<br>cificPart | The scheme specific part of the URL is improperly formatted for the specified scheme.  |

#### [-ORBDefaultInitRef <ObjectURL>]

The ORB default initial reference argument, -ORBDefaultInitRef, assists in the resolution of initial references not explicitly specified with -ORBInitRef. This argument provides functionality similar to that of the list of IIOP Listeners address that is provided to the current Tobj Bootstrap object.

Unlike the <code>-ORBInitRef</code> argument, <code>-ORBDefaultInitRef</code> requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example, if the following was specified as the default initial reference argument:

#### -ORBDefaultInitRef corbaloc:555objs.com

A call to  $\c ORB::resolve\_initial\_references$  ("NotificationService") to obtain the initial reference for the service would result in the new URL:

#### corbaloc:555objs.com/NotificationService

The implementation of the <code>ORB::resolve\_initial\_references</code> operation would take the newly constructed URL and call <code>CORBA::ORB::string\_to\_object</code> to obtain the initial reference for the service.

The URL specified as the value of the <code>-ORBDefaultInitRef</code> argument can contain more than a single location. This is the similar to the functionality provided for the list of locations to be used by the <code>Tobj\_Bootstrap</code> object. In this situation, the ORB will process the locations in the URL based on the syntax rules for the URL. For example, if the following was specified as the default initial reference argument:

#### -ORBDefaultInitRef corbaloc:555objs.com,555Backup.com

A call to ORB::resolve\_initial\_references("NameService") to obtain the initial reference for the service would result in one of the following new URLs:

corbaloc:555objs.com/NameService Or corbaloc:555Backup.com/NameService
The resulting URL would then be passed to CORBA::ORB::string\_to\_object in order to obtain the initial reference for the service.

#### [-ORBminCrypto [0 | 40 | 56 | 128]]

When establishing a network link, this is the minimum level of encryption required. Zero (0) means no encryption, while 40, 56, and 128 specify the length (in bits) of the encryption key. If this minimum level of encryption cannot be met, link establishment will fail.

The default is 0.

#### [-ORBmaxCrypto [0 | 40 | 56 | 128]]

When establishing a network link, this is the maximum level of encryption allowed. Zero (0) means no encryption, while 40, 56, and 128 specify the length (in bits) of the encryption key. The default is whatever capability is specified by the license. The - ORBmaxCrypto or -ORBmaxCrypto options are available only if either the International or U.S Canada Oracle Tuxedo Security Add-on Package is installed.

#### [-ORBmutualAuth]

Specifies that certificate-based authentication must be enabled when accepting an SSL connection from a remote application.

The -ORBmutualAuth option is available only if either the International or U.S\_Canada Oracle Tuxedo Security Add-on Package is installed.

#### [-ORBpeerValidate {detect | warn | none}]

Determines how the Oracle Tuxedo CORBA ORB will behave when a digital certificate for a peer of an outbound connection initiated by the Oracle Tuxedo ORB is received as part of the Secure Socket Layer (SSL) protocol handshake. The validation is only performed by the initiator of a secure connection and confirms that the peer server is actually located at the same network address specified by the domain name in the server's digital certificate. This validation is not technically part of the SSL protocol, but is similar to the same check done in web browsers.

A value of detect causes an Oracle Tuxedo CORBA ORB to verify that the host specified in the object reference used to make the connection matches the domain name specified in the peer's digital certificate. If the comparison fails, the ORB refuses to authenticate the peer and drops the connection. This check protects against man-in-the-middle attacks.

A value of warn causes an Oracle Tuxedo CORBA ORB to verify that the host specified in the object reference used to make the connection matches the domain name specified in the peer's digital certificate. If the comparison fails, the ORB logs a message to the user log, but continues processing the connection.

A value of none causes an Oracle Tuxedo CORBA ORB not to perform the peer validation and will continue the processing of the connection.

The -ORBpeerValidate option is available only if either the International or U.S\_Canada Oracle Tuxedo Security Add-on Package is installed.

If not specified, the default is detect.



#### [-ORBport port-number]

Specifies the network address to be used by the ORB to accept connections from remote CORBA clients. Typically, a system administrator assigns the port number for the client from the "user" range of port numbers rather from the dynamic range. This keeps the joint client/servers from using conflicting ports.

This parameter is required in order for the Oracle Tuxedo CORBA ORB to create persistent object references. Persistent objects references must be served on the same port after that is contained in the object reference, even if the ORB has been restarted. For transient object references, any port is sufficient and can be obtained by the ORB dynamically.

The port-number is the TCP port number at which the Oracle Tuxedo CORBA ORB process listens for incoming requests. The port-number can be a number between 0 and 65535.

#### [-ORBsecurePort port-number]

Specifies the port number that the IIOP Listener/Handler must use to listen for secure connections using the Secure Socket Layer protocol. If the command-line option is specified without a port number, then the OMG assigned port number 684 will be used for SSL connections.

The port-number is the TCP port number at which the Oracle Tuxedo CORBA ORB process listens for incoming requests. The port-number can be a number between 0 and 65535.

An administrator can configure to only allow secure connections into the Oracle Tuxedo CORBA ORB by setting port numbers specified by the <code>-ORBport</code> and <code>-ORBsecurePort</code> to the same value.

The -ORBsecurePort option is available only if either the International or U.S\_Canada Oracle Tuxedo Security Add-on Package is installed.

## 14.11.2.5 Portability

The Oracle Tuxedo CORBA ORB is supported as an Oracle Tuxedo-supplied client or server on UNIX and Microsoft Windows operating systems. It is also supported as an Oracle Tuxedo-supplied client on the Windows XP operating systems.

## 14.11.2.6 Interoperability

The Oracle Tuxedo CORBA ORB will interoperate with any IIOP compliant ORB that supports the 1.0, 1.1, or 1.2 version of the GIOP protocol over a TCP/IP connection. In addition, the Oracle Tuxedo CORBA ORB will interoperate with any IIOP-SSL compliant ORB that supports the use of the TAG\_SSL\_SEC\_TRANS tagged component in object references and version 3 of the Secure Socket Layer protocol.

## 14.11.2.7 Examples

#### C++ code example

ChatClient -ORBid BEA\_IIOP -ORBport 2100 -ORBDefaultInitRef corbaloc:piglet:1900 -ORBInitRef TraderService=corbaloc:owl:2530

- -ORBsecurePort 2100
- -ORBminCrypto 40



```
-ORBmaxCrypto 128 TechTopics
```

#### Java code example

```
java -DORBDefaultInitRef=corbalocs:piglet:1900
....-DORBInitRef=TraderService=corbaloc:owl:2530
-Dorg.omg.CORBA.ORBPort=1948
-classpath=%CLASSPATH% client
```

### 14.11.2.8 See Also

ISL

# 14.12 Policy Member Functions

A policy is an object used to communicate certain choices to an ORB regarding its operation. This information is accessed in a structured manner using interfaces derived from the Policy interface defined in the CORBA module.



These CORBA::Policy operations and structures are not usually needed by programmers. The derived interfaces usually contain the information relevant to specifications. A policy object can be constructed by a specific factory or by using the CORBA::create policy operation.

The mapping of this object to C++ is as follows:

```
class CORBA
{
  class Policy
{
  public:
  copy();
  void destroy();
  }; //Policy
  typedef sequence<Policy>PolicyList;
  }; // CORBA
```

PolicyList is used the same as any other C++ sequence mapping. For a discussion of sequence usage, see Sequences.

- CORBA:Policy::copy
- CORBA::Policy::destroy



See Also:

POA Policy and CORBA::ORB::create\_policy .

# 14.12.1 CORBA:Policy::copy

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

### 14.12.1.1 Synopsis

Copies the policy object.

### 14.12.1.2 C++ Binding

CORBA::Policy::copy();

### 14.12.1.3 Argument

None.

# 14.12.1.4 Description

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain or object.

Note:

This function is supported only for a joint client/server.

### 14.12.1.5 Return Values

None.

# 14.12.2 CORBA::Policy::destroy

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description



#### Return Values

### 14.12.2.1 Synopsis

Destroys the policy object.

# 14.12.2.2 C++ Binding

```
void CORBA::Policy::destroy();
```

### 14.12.2.3 Argument

None.

## 14.12.2.4 Exceptions

If the policy object determines that it cannot be destroyed, the CORBA::NO\_PERMISSION exception is raised.

## 14.12.2.5 Description

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.



This function is supported only for a joint client/server.

### 14.12.2.6 Return Values

None.

# 14.13 PortableServer Member Functions

The mapping of the PortableServer member functions to C++ is as follows:

```
// C++
class PortableServer
{
public:
class LifespanPolicy;
class IdAssignmentPolicy;
class POA::find_POA
class reference_to_id
class POAManager;
class POA;
class Current;
class virtual ObjectId
class ServantBase
};
```

#### ObjectId

A value that is used by the POA and by the user-supplied implementation to identify a particular abstract CORBA object. ObjectId values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. ObjectId values are hidden from clients, encapsulated by references. ObjectIds have no standard form; they are managed by the POA as uninterpreted octet sequences.

The following sections describe the remaining classes.

- PortableServer::POA::activate\_object
- PortableServer::POA::activate\_object\_with\_id
- PortableServer::POA::create\_id\_assignment\_policy
- PortableServer::POA::create\_lifespan\_policy
- PortableServer::POA::create\_POA
- PortableServer::POA::create\_reference
- PortableServer::POA::create\_reference\_with\_id
- PortableServer::POA::deactivate\_object
- PortableServer::POA::destroy
- PortableServer::POA::find\_POA
- PortableServer::POA::reference\_to\_id
- PortableServer::POA::the\_POAManager
- PortableServer::ServantBase::\_default\_POA

# 14.13.1 PortableServer::POA::activate object

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- Example

### 14.13.1.1 Synopsis

Explicitly activates an individual object.

### 14.13.1.2 C++ Binding

```
ObjectId * activate_object (
Servant p servant);
```

### 14.13.1.3 Argument

#### p\_servant

An instance of the C++ implementation class for the interface.

### 14.13.1.4 Exceptions

If the specified servant is already in the Active Object Map, the ServantAlreadyActive exception is raised.



Other exceptions can occur if the POA uses unsupported policies.

### 14.13.1.5 Description

This operation explicitly activates an individual object by generating an ObjectId and entering the ObjectId and the specified servant in the Active Object Map.

Note

This function is supported only for a joint client/server.

### 14.13.1.6 Return Values

If the function succeeds, the ObjectId is returned.

## 14.13.1.7 Example

In the following example, the first struct creates a servant by a user-defined constructor. The second struct tells the POA that the servant can be used to handle requests on an object. The POA returns the <code>ObjectId</code> it has created for the object. The third statement assumes that the POA has the <code>IMPLICIT\_ACTIVATION</code> policy (the only supported policy in version 4.2 of the Oracle Tuxedo software) and returns a reference to the object. That reference can then be handed to a client for invocations. When the client invokes on the reference, the request is returned to the servant just created.

```
MyFooServant* afoo = new MyFooServant(poa,27);
PortableServer::ObjectId_var oid =
poa->activate_object(afoo);
Foo_var foo = afoo->_this();
```

# 14.13.2 PortableServer::POA::activate\_object\_with\_id

- Synopsis
- C++ Binding
- Argument

- Exceptions
- Description
- Return Values
- Example

### 14.13.2.1 Synopsis

Activates an individual object with a specified ObjectId.

### 14.13.2.2 C++ Binding

```
void activate_object_with_id (
const ObjectId & id,
Servant p servant);
```

## 14.13.2.3 Argument

#### id

ObjectId that identifies the object on which that operation was invoked.

#### p servant

An instance of the C++ implementation class for the interface.

# 14.13.2.4 Exceptions

The <code>ObjectAlreadyActive</code> exception is raised if the CORBA object denoted by the <code>ObjectId</code> value is already active in this POA.

The ServantAlreadyActive exception is raised if the servant is already in the Active Object Map.



Other exceptions can occur if the POA uses unsupported policies.

The BAD\_PARAM system exception may be raised if the POA has the SYSTEM\_ID policy and it detects that the ObjectId value was not generated by the system or for this POA. An ORB is not required to detect all such invalid ObjectId values. However, a portable application must not invoke activate\_object\_with\_id on a POA if the POA has the SYSTEM\_ID policy with an ObjectId value that was not previously generated by the system for that POA, or, if the POA also has the PERSISTENT policy, for a previous instantiation of the same POA.

## 14.13.2.5 Description

This operation enters an association between the specified ObjectId and the specified servant in the Active Object Map.



This function is supported only for a joint client/server.

### 14.13.2.6 Return Values

None.

### 14.13.2.7 Example

```
MyFooServant* afoo = new MyFooServant(poa, 27);
PortableServer::ObjectId_var oid =
PortableServer::string_to_ObjectId("myLittleFoo");
poa->activate_object_with_id(oid.in(), afoo);
Foo_var foo = afoo->_this();
```

# 14.13.3 PortableServer::POA::create\_id\_assignment\_policy

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

## 14.13.3.1 Synopsis

Obtains an object with the IdAssignmentPolicy interface so the user can pass the object to the POA::create POA operation.

## 14.13.3.2 C++ Binding

```
IdAssignmentPolicy_ptr
PortableServer::POA::create_id_assignment_policy (
PortableServer:: IdAssignmentPolicyValue value)
```

## 14.13.3.3 Argument

#### value

A value of either PortableServer:: USER\_ID, indicating ObjectIds are assigned only by the application, or PortableServer:: SYSTEM\_ID, indicating ObjectIds are assigned only by the system.

### 14.13.3.4 Description

The POA::create\_id\_assignment\_policy operation obtains objects with the IdAssignmentPolicy interface. When passed to the POA::create\_POA operation, this policy specifies whether ObjectIds in the created POA are generated by the application or by the ORB. The following values can be supplied:

- PortableServer:: USER\_ID— objects created with that POA are assigned ObjectIds only by the application.
- PortableServer:: SYSTEM\_ID— objects created with that POA are assigned ObjectIds only by the POA. If the POA also has the PERSISTENT LifespanPolicy, assigned ObjectIds must be unique across all instantiations of the same POA.

If no IdAssignmentPolicy is specified at POA creation, the default is SYSTEM ID.



This function is supported only for a joint client/server.

### 14.13.3.5 Return Values

Returns an Id Assignment policy.

# 14.13.4 PortableServer::POA::create\_lifespan\_policy

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

## 14.13.4.1 Synopsis

Obtains an object with the LifespanPolicy interface so the user can pass the object to the POA::create\_POA operation.

## 14.13.4.2 C++ Binding

```
LifespanPolicy_ptr
PortableServer::POA::create_lifespan_policy (
PortableServer::LifespanPolicyPolicyValue value)
```

### 14.13.4.3 Argument

#### value

A value of either PortableServer:: USER\_ID, indicating ObjectIds are assigned only by the application, or PortableServer:: SYSTEM\_ID, indicating ObjectIds are assigned only by the system.

### 14.13.4.4 Description

Objects with the LifespanPolicy interface are obtained using the POA::create\_lifespan\_policy operation and passed to the POA::create\_POA operation to specify the lifespan of the objects implemented in the created POA. The following values can be supplied.

- TRANSIENT—the objects implemented in the POA cannot outlive the process in which they
  are first created. Once the POA is deactivated, use of any object references generated
  from it will result in an OBJECT\_NOT\_EXIST exception.
- PERSISTENT—the objects implemented in the POA can outlive the process in which they
  are first created.
- Persistent objects have a POA associated with them (the POA which created them). When
  the ORB receives a request on a persistent object, it first searches for the matching POA,
  based on the names of the POA and all of its ancestors.
- Administrative action beyond the scope of this specification may be necessary to inform
  the ORB's location service of the creation and eventual termination of existence of this
  POA, and optionally to arrange for on-demand activation of a process implementing this
  POA.
- POA names must be unique within their enclosing scope (the parent POA). A portable
  program can assume that POA names used in other processes will not conflict with its own
  POA names. A conforming CORBA implementation will provide a method for ensuring this
  property.

If no LifespanPolicy object is passed to POA::create\_POA, the lifespan policy defaults to TRANSIENT.



This function is supported only for a joint client/server.

### 14.13.4.5 Return Values

Returns a LifespanPolicy.

# 14.13.5 PortableServer::POA::create\_POA

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- Examples

# 14.13.5.1 Synopsis

Creates a new POA as a child of the target POA.

### 14.13.5.2 C++ Binding

```
POA_ptr PortableServer::create_POA (
const char * adapter name,
```

POAManager\_ptr a\_POAManager,
const CORBA::PolicyList & policies)

### 14.13.5.3 Arguments

#### adapter name

The name of the POA to be created.

#### a POAManager

Either a NULL value, indicating that a new POAManager is to be created and associated with the new POA, or a pointer to an existing POAManager.

#### policies

Policy objects to be associated with the new POA.

### 14.13.5.4 Exceptions

#### AdapterAlreadyExists

Raised if the target POA already has a child POA with the specified name.

#### InvalidPolicy

Raised if any of the policy objects specified are not valid for the ORB implementation, if conflicting policy objects are specified, or if any of the specified policy objects require prior administrative action that has not been performed. This exception contains the index in the policy parameter value of the first offending policy object.

#### IMP LIMIT

Raised if the program tries to create a POA with a LifespanPolicy of PERSISTENT without having set a port, as described in the operation CORBA::ORB\_init.

## 14.13.5.5 Description

This operation creates a new POA as a child of the target POA. The specified name, which must be unique, identifies the new POA with respect to other POAs with the same parent POA.

If the a\_POAManager parameter is NULL, a new PortableServer::POAManager object is created and associated with the new POA. Otherwise, the specified POAManager object is associated with the new POA. The POAManager object can be obtained using the attribute name the\_POAManager.

The specified policy objects are associated with the POA and are used to control its behavior. The policy objects are effectively copied before this operation returns, so the application is free to destroy them while the POA is in use. Policies are *not* inherited from the parent POA.



This function is supported only for joint client/servers.

### 14.13.5.6 Return Values

Returns a pointer to the POA that was created.

### 14.13.5.7 Examples

### **Example 1**

In this example, the child POA would use the same manager as the parent POA; the child POA would then have the same state as the parent (that is, it would be active if the parent is active).

```
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
rootPOA->create_POA("my_little_poa",
rootPOA->the_POAManager, policies);
```

#### Example 2

In this example, a new POA is created as a child of the root POA.

```
CORBA::PolicyList policies(2);
policies.length (1);
policies[0] = rootPOA->create_lifespan_policy(
PortableServer::LifespanPolicy::TRANSIENT);
PortableServer::POA_ptr poa =
rootPOA->create_POA("my_little_poa",
PortableServer::POAManager::_nil(), policies);
```

# 14.13.6 PortableServer::POA::create\_reference

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values

## 14.13.6.1 Synopsis

Creates an object reference that encapsulates a POA-generated ObjectId value and the specified interface repository ID.

## 14.13.6.2 C++ Binding

### 14.13.6.3 Argument

intf

The interface repository ID.

### 14.13.6.4 Exceptions

This operation requires the LifespanPolicy to have the value SYSTEM\_ID; if not present, the PortableServer::WrongPolicy exception is raised.

## 14.13.6.5 Description

This create\_reference operation creates an object reference that encapsulates a POA-generated <code>ObjectId</code> value and the specified interface repository ID. This operation collects the necessary information to constitute the reference from information associated with the POA and from parameters to the operation. This operation only creates a reference; it does not associate the reference with an active servant. The resulting reference may be passed to clients, so that subsequent requests on those references return to the POA using the <code>ObjectId</code> generated. The generated <code>ObjectId</code> value may be obtained by invoking <code>POA::reference to id</code> with the created reference.

Note: This function is supported only for a joint client/server.

### 14.13.6.6 Return Values

Returns a pointer to the object.

# 14.13.7 PortableServer::POA::create\_reference\_with\_id

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- Example

## 14.13.7.1 Synopsis

Creates an object reference that encapsulates the specified <code>ObjectId</code> and interface repository ID values.

# 14.13.7.2 C++ Binding

```
CORBA::Object_ptr create_reference_with_id (
const ObjectId & oid,
const char * intf)
```

### 14.13.7.3 Arguments

#### oid

ObjectId that identifies the object on which that operation was invoked.

#### intf

The interface repository ID.

### 14.13.7.4 Exceptions

If the POA has a LifespanPolicy with value SYSTEM\_ID and it detects that the ObjectId value was not generated by the system or for this POA, the operation will raise the BAD\_PARAM system exception.

### 14.13.7.5 Description

The create\_reference operation creates an object reference that encapsulates the specified <code>ObjectId</code> and interface repository ID values. This operation collects the necessary information to constitute the reference from information associated with the POA and from parameters to the operation. This operation only creates a reference; it does not associate the reference with an active servant. The resulting reference may be passed to clients, so that subsequent requests on those references cause the invocation to be returned to the same POA with <code>ObjectId</code> specified.



This function is supported only for a joint client/server.

### 14.13.7.6 Return Values

Returns Object ptr.

## 14.13.7.7 Example

```
PortableServer::ObjectId_var oid =
PortableServer::string_to_ObjectId("myLittleFoo");
CORBA::Object_var obj = poa->create_reference_with_id(oid.in(), "IDL:Foo:1.0");
Foo_var foo = Foo::_narrow(obj);
```

# 14.13.8 PortableServer::POA::deactivate\_object

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description



#### Return Values

# 14.13.8.1 Synopsis

Removes the ObjectId from the Active Object Map.

## 14.13.8.2 C++ Binding

```
void deactivate_object (
const ObjectId & oid)
```

## 14.13.8.3 Argument

#### oid

ObjectId that identifies the object.

### 14.13.8.4 Exceptions

If there is no active object associated with the specified <code>ObjectId</code>, the operation raises an <code>ObjectNotActive</code> exception.

## 14.13.8.5 Description

This operation causes the association of the <code>ObjectId</code> specified by the <code>oid</code> parameter and its servant to be removed from the Active Object Map.



This function is supported only for a joint client/server.

### 14.13.8.6 Return Values

None.

# 14.13.9 PortableServer::POA::destroy

- Synopsis
- · C++ Binding
- Arguments
- Description
- Return Values

# 14.13.9.1 Synopsis

Destroys the POA and all descendant POAs.

### 14.13.9.2 C++ Binding

```
void destroy (
CORBA::Boolean etherealize_objects,
CORBA::Boolean wait_for_completion)
```

### 14.13.9.3 Arguments

#### etherealize objects

This argument must be FALSE for this release of Oracle Tuxedo.

#### wait for completion

This argument indicates whether or not the operation must return immediately.

## 14.13.9.4 Description

This operation destroys the POA and all descendant POAs. The POA with its name may be recreated later in the same process. (This differs from the POAManager::deactivate operation, which does not allow a recreation of its associated POA in the same process.)

When a POA is destroyed, any requests that have started execution continue to completion. Any requests that have not started execution are processed as if they were newly arrived and there is no POA; that is, they are rejected and the <code>OBJECT NON EXIST</code> exception is raised.

If the wait\_for\_completion parameter is TRUE, the destroy operation returns only after all requests in process have completed and all invocations of etherealize have completed. Otherwise, the destroy operation returns after destroying the POAs.

### Note:

This release of Oracle Tuxedo does not support multithreading. Hence, wait\_for\_completion must not be TRUE if the call is made in the context of an object invocation. That is, the POA cannot start destroying itself if it is currently executing. This function is supported only for a joint client/server.

### 14.13.9.5 Return Values

None.

## 14.13.10 PortableServer::POA::find POA

- Synopsis
- C++ Binding
- Argument
- Exception
- Description
- Return Values



### 14.13.10.1 Synopsis

Returns a reference to a child POA with a given name.

## 14.13.10.2 C++ Binding

void find POA( in string adapter name, in boolean activate it);

### 14.13.10.3 Argument

#### adapter name

A reference to the target POA.

#### active it

In this version of Oracle Tuxedo, this parameter must be FALSE.

### 14.13.10.4 Exception

#### AdapterNonExistent

This exception is raised if the POA does not exist.

### 14.13.10.5 Description

If the POA has a child POA with the specified name, that child POA is returned. If a child POA with the specified name does not exist and the value of the activate\_it parameter is FALSE, the AdapterNonExistent exception is raised.

### 14.13.10.6 Return Values

None.

# 14.13.11 PortableServer::POA::reference\_to\_id

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values

## 14.13.11.1 Synopsis

Returns the ObjectId value encapsulated by the specified reference.

# 14.13.11.2 C++ Binding

ObjectId reference\_to\_id(in Object reference);

### 14.13.11.3 Argument

#### reference

Specifies the reference to the object.

### 14.13.11.4 Exceptions

#### WrongAdapter

This exception is raised if the reference was not created by that POA.

### 14.13.11.5 Description

This operation returns the <code>ObjectId</code> value encapsulated by the specified <code>reference</code>. This operation is valid only if the reference was created by the POA on which the operation is being performed. The object denoted by the reference does not have to be active for this operation to succeed.



This function is supported only for a joint client/server.

# 14.13.11.6 Return Values

Returns the ObjectId value encapsulated by the specified reference.

# 14.13.12 PortableServer::POA::the\_POAManager

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example

## 14.13.12.1 Synopsis

Identifies the POA manager associated with the POA.

## 14.13.12.2 C++ Binding

POAManager ptr the POAManager ();

## 14.13.12.3 Argument

None.

### 14.13.12.4 Description

This read-only attribute identifies the POA manager associated with the POA.



This function is supported only for a joint client/server.

### 14.13.12.5 Return Values

None.

## 14.13.12.6 Example

```
poa->the_POAManager()->activate();
```

This statement will set the state of the POAManager for the given POA to active, which is required if the POA is to accept requests. Note that if the POA has a parent, that is, it is not the root POA, all of its parent's POAManagers must also be in the active state for this statement to have any effect.

# 14.13.13 PortableServer::ServantBase::\_default\_POA

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

# 14.13.13.1 Synopsis

Returns an object reference to the POA associated with the servant.

## 14.13.13.2 C++ Binding

```
class PortableServer
{
  class ServantBase
  {
  public:
    virtual POA_ptr _default_POA();
  }
}
```

# 14.13.13.3 Argument

None.

### 14.13.13.4 Description

All C++ Servants inherit from PortableServer::ServantBase, so they all inherit the \_default\_POA function. In this version of Oracle Tuxedo there is usually no reason to use default\_POA.

The default implementation of this function returns an object reference to the root POA of the default ORB in this process—the same as the return value of an invocation of ORB::resolve\_initial\_references("RootPOA"). A C++ servant can override this definition to return the POA of its choice, if desired.



This function is supported only for joint client/servers.

### 14.13.13.5 Return Values

The default POA associated with the servant.

### 14.14 POA Current Member Functions

The PortableServer::Current interface, derived from CORBA::Current, provides method implementations with access to the identity of the object on which the method was invoked.

- PortableServer::Current::get object id
- PortableServer::Current::get\_POA

# 14.14.1 PortableServer::Current::get\_object\_id

- Synopsis
- C++ Binding
- Arguments
- Exception
- Description
- Return Values

## 14.14.1.1 Synopsis

Returns the ObjectId identifying the object in whose context it is called.

# 14.14.1.2 C++ Binding

```
ObjectId * get_object_id ();
```

## 14.14.1.3 Arguments

None.

### 14.14.1.4 Exception

If called outside the context of a POA-dispatched operation, a PortableServer::NoContext exception is raised.

## 14.14.1.5 Description

This operation returns the PortableServer::ObjectId identifying the object in whose context it is called.



This function is supported only for a joint client/server.

#### 14.14.1.6 Return Values

This operation returns the ObjectId identifying the object in whose context it is called.

# 14.14.2 PortableServer::Current::get POA

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values

# 14.14.2.1 Synopsis

Returns a reference to the POA implementing the object in whose context it is called.

# 14.14.2.2 C++ Binding

```
POA_ptr get_POA ();
```

## 14.14.2.3 Argument

None.

## 14.14.2.4 Exceptions

If this operation is called outside the context of a POA-dispatched operation, a PortableServer::NoContext exception is raised.

### 14.14.2.5 Description

This operation returns a reference to the POA implementing the object in whose context it is called.



This function is supported only for a joint client/server.

### 14.14.2.6 Return Values

This operation returns a reference to the POA implementing the object in whose context it is called.

# 14.15 POAManager Member Functions

Each POA object has an associated POAManager object. A POAManager may be associated with one or more POA objects. A POAManager encapsulates the processing state of the POAs with which it is associated. Using operations on the POA manager, an application can cause requests for those POAs to be queued or discarded, and can cause the POAs to be deactivated.

POA managers are created and destroyed implicitly. Unless an explicit POAManager object is provided at POA creation time, a POAManager is created when a POA is created and is automatically associated with that POA. A POAManager object is implicitly destroyed when all of its associated POAs have been destroyed.

A POAmanager has four possible processing states: *active*, *inactive*, *holding*, and *discarding*. The processing state determines the capabilities of the associated POAs and the disposition of requests received by those POAs.

A POAmanager is created in the holding state. In that state, any invocations on its POA are queued until the POA manager enters the active state. This version of Oracle Tuxedo supports only the ability to enter active and inactive states. That is, this version does not support the ability to return to holding state or to enter discarding state.

- PortableServer::POAManager::activate
- PortableServer::POAManager::deactivate

# 14.15.1 PortableServer::POAManager::activate

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values



### 14.15.1.1 Synopsis

Changes the state of the POAManager to active.

### 14.15.1.2 C++ Binding

void activate();

### 14.15.1.3 Argument

None.

## 14.15.1.4 Exceptions

If this operation is issued while the POAmanager is in the *inactive* state, the PortableServer::POAManager::AdapterInactive exception is raised.

## 14.15.1.5 Description

This operation changes the state of the POAManager to *active*. Entering the *active* state enables the associated POAs to process requests.



All parent POAs must also have POAManagers in the active state for this POA to process requests.

This function is supported only for a joint client/server.

## 14.15.1.6 Return Values

None.

# 14.15.2 PortableServer::POAManager::deactivate

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values

# 14.15.2.1 Synopsis

Changes the state of the POA manager to *inactive*.

### 14.15.2.2 C++ Binding

```
void deactivate (
CORBA::Boolean etherealize_objects,
CORBA::Boolean wait for completion);
```

### 14.15.2.3 Argument

#### etherealize objects

For BEA WebLogic Enterprise 4.2 software and later software and Oracle Tuxedo 8.0 and later software, this argument must always be set to FALSE.

#### wait for completion

If this argument is TRUE, the deactivate operation returns only after all requests in process have completed. If this argument is FALSE, the deactivate operation returns after changing the state of the associated POAs.

### 14.15.2.4 Exceptions

If issued while the POA manager is in the *inactive* state, the

PortableServer::POAManager::AdapterInactive exception is raised.

### 14.15.2.5 Description

This operation changes the state of the POAManager to *inactive*. Entering the inactive state causes the associated POAs to reject requests that have not begun to be executed, as well as any new requests.



This release of Oracle Tuxedo does not support multithreading. Hence, wait\_for\_completion must not be TRUE if the call is made in the context of an object invocation. That is, the POAManager cannot be set to inactive state if it is currently executing.

This function is supported only for a joint client/server.

### 14.15.2.6 Return Values

None.

# 14.16 POA Policy Member Objects

Interfaces derived from CORBA::Policy are used with the POA::create\_POA operation to specify policies that apply to a POA. Policy objects are created using factory operations on any preexisting POA, such as the root POA. Policy objects are specified when a POA is created. Policies may not be changed on an existing POA. Policies are *not* inherited from the parent POA.

PortableServer::LifespanPolicy

PortableServer::IdAssignmentPolicy

# 14.16.1 PortableServer::LifespanPolicy

- Synopsis
- Description
- Exceptions

### 14.16.1.1 Synopsis

Specifies the life span of objects to the create POA operation.

### 14.16.1.2 Description

Objects with the LifespanPolicy interface are obtained using the POA::create\_lifespan\_policy operation and are passed to the POA::create\_POA operation to specify the life span of the objects implemented in the created POA. The following values can be supplied:

- TRANSIENT—the objects implemented in the POA cannot outlive the process in which they
  are first created.
- PERSISTENT—the objects implemented in the POA can outlive the process in which they
  are first created.

Persistent objects have a POA associated with them (the POA that created them). When the ORB receives a request on a persistent object, it searches for the matching POA, based on the names of the POA and all of its ancestors.

POA names must be unique within their enclosing scope (the parent POA). A portable program can assume that POA names used in other processes will not conflict with its own POA names.

If no Lifespan Policy object is passed to create POA, the lifespan policy defaults to TRANSIENT.

Note: This function is supported only for a joint client/server.

### 14.16.1.3 Exceptions

None.

# 14.16.2 PortableServer::IdAssignmentPolicy

- Synopsis
- Description

## 14.16.2.1 Synopsis

Specifies whether <code>ObjectIds</code> in the created POA are generated by the application or by the ORB.

## 14.16.2.2 Description

Objects with the IdAssignmentPolicy interface are obtained using the POA::create id assignment policy operation and are passed to the POA::create POA

operation to specify whether ObjectIds in the created POA are generated by the application or by the ORB. The following values can be supplied:

- USER ID objects created with that POA are assigned ObjectIds only by the application.
- SYSTEM\_ID— objects created with that POA are assigned ObjectIds only by the POA. If
  the POA also has the PERSISTENT policy, assigned ObjectIds must be unique across all
  instantiations of the same POA

If no IdAssignmentPolicy is specified at POA creation, the default is SYSTEM ID.

Note: This function is supported only for a joint client/server.

# 14.17 Request Member Functions

The mapping of these member functions to C++ is as follows:

```
// C++
class Request
public:
Object_ptr target() const;
const char *operation() const;
NamedValue ptr result();
NVList ptr arguments();
Environment_ptr env();
ExceptionList ptr exceptions();
ContextList ptr contexts();
void ctx(Context ptr);
Context ptr ctx() const
// argument manipulation helper functions
Any &add in arg();
Any &add in arg(const char* name);
Any &add inout arg():
Any &add_inout_arg(const char* name);
Any &add out arg():
Any &add out arg(const char* name);
void set return type(TypeCode ptr tc);
Any &return value();
void invoke();
void send oneway();
void send deferred();
void get response();
Boolean poll response();
};
```

#### Note:

The add\_\*\_arg, set\_return\_type, and return\_value member functions are added as shortcuts for using the attribute-based accessors.

The following sections describe these member functions.

- CORBA::Request::arguments
- CORBA::Request::ctx(Context\_ptr)
- CORBA::Request::get\_response
- CORBA::Request::invoke
- CORBA::Request::operation
- CORBA::Request::poll\_response
- CORBA::Request::result
- CORBA::Request::env
- CORBA::Request::ctx
- CORBA::Request::contexts
- CORBA::Request::exceptions
- CORBA::Request::target
- CORBA::Request::send\_deferred
- CORBA::Request::send\_oneway

## 14.17.1 CORBA::Request::arguments

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

## 14.17.1.1 Synopsis

Retrieves the argument list for the request.

## 14.17.1.2 C++ Binding

```
CORBA::NVList ptr CORBA::Request::arguments () const;
```

## 14.17.1.3 Arguments

None.

# 14.17.1.4 Description

This member function retrieves the argument list for the request. The arguments can be input, output, or both.

#### 14.17.1.5 Return Values

If the function succeeds, the value returned is a pointer to the list of arguments to the operation for the request. The returned argument list is owned by the Request object reference and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.2 CORBA::Request::ctx(Context\_ptr)

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- See Also

# 14.17.2.1 Synopsis

Sets the Context object for the operation.

### 14.17.2.2 C++ Binding

```
void CORBA::Request::ctx (
CORBA::Context ptr CtxObject);
```

# 14.17.2.3 Argument

#### CtxObject

The new value to which to set the Context object.

### 14.17.2.4 Description

This member function sets the Context object for the operation.

### 14.17.2.5 Return Values

None.

## 14.17.2.6 See Also

```
CORBA::Request::ctx()
```

# 14.17.3 CORBA::Request::get\_response

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also



## 14.17.3.1 Synopsis

Retrieves the response of a specific deferred synchronous request.

## 14.17.3.2 C++ Binding

```
void CORBA::Request::get response ();
```

### 14.17.3.3 Arguments

None.

## 14.17.3.4 Description

This member function retrieves the response of a specific request; it is used after a call to the CORBA::Request::send\_deferred function or the CORBA::Request::send\_multiple\_requests function. If the request has not completed, the CORBA::Request::get\_response function blocks until it does complete.

### 14.17.3.5 Return Values

None.

### 14.17.3.6 See Also

```
CORBA::Request::send deferred
```

# 14.17.4 CORBA::Request::invoke

- Synopsis
- · C++ Binding
- Arguments
- Description
- Return Values

# 14.17.4.1 Synopsis

Performs an invoke on the operation specified in the request.

## 14.17.4.2 C++ Binding

```
void CORBA::Request::invoke ();
```

## 14.17.4.3 Arguments

None.

### 14.17.4.4 Description

This member function calls the Object Request Broker (ORB) to send the request to the appropriate server application.

#### 14.17.4.5 Return Values

None.

# 14.17.5 CORBA::Request::operation

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

## 14.17.5.1 Synopsis

Retrieves the operation intended for the request.

### 14.17.5.2 C++ Binding

```
const char * CORBA::Request::operation () const;
```

# 14.17.5.3 Arguments

None.

# 14.17.5.4 Description

This member function retrieves the operation intended for the request.

### 14.17.5.5 Return Values

If the function succeeds, the value returned is a pointer to the operation intended for the object; the value can be 0 (zero). The memory returned is owned by the Request object and must not be freed.

If the function does not succeed, an exception is thrown.

# 14.17.6 CORBA::Request::poll\_response

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values



#### See Also

# 14.17.6.1 Synopsis

Determines whether a deferred synchronous request has completed.

# 14.17.6.2 C++ Binding

```
CORBA::Boolean CORBA::Request::poll response ();
```

### 14.17.6.3 Arguments

None.

## 14.17.6.4 Description

This member function determines whether the request has completed and returns immediately. You can use this call to check the state of the request. This member function can also be used to determine whether a call to CORBA::Request::get response will block.

#### 14.17.6.5 Return Values

If the function succeeds, the value returned is <code>CORBA\_TRUE</code> if the response has already completed, and <code>CORBA\_FALSE</code> if the response has not yet completed.

If the function does not succeed, an exception is thrown.

### 14.17.6.6 See Also

```
CORBA::ORB::get_next_response

CORBA::ORB::poll_next_response

CORBA::ORB::send_multiple_requests

CORBA::Request::get_response

CORBA::Request::send_deferred
```

# 14.17.7 CORBA::Request::result

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.17.7.1 Synopsis

Retrieves the result of the request.

## 14.17.7.2 C++ Binding

```
CORBA::NamedValue ptr CORBA::Request::result ();
```

### 14.17.7.3 Arguments

None.

### 14.17.7.4 Description

This member function retrieves the result of the request.

#### 14.17.7.5 Return Values

If the function succeeds, the value returned is a pointer to the result of the operation. The returned result is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.8 CORBA::Request::env

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.17.8.1 Synopsis

Retrieves the environment of the request.

## 14.17.8.2 C++ Binding

```
CORBA::Environment_ptr CORBA::Request::env ();
```

# 14.17.8.3 Arguments

None.

### 14.17.8.4 Description

This member function retrieves the environment of the request.

#### 14.17.8.5 Return Values

If the function succeeds, the value returned is a pointer to the environment of the operation. The returned environment is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.9 CORBA::Request::ctx

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.17.9.1 Synopsis

Retrieves the context of the request.

## 14.17.9.2 C++ Binding

```
CORBA::context ptr CORBA::Request::ctx ();
```

### 14.17.9.3 Arguments

None.

## 14.17.9.4 Description

This member function retrieves the context of the request.

#### 14.17.9.5 Return Values

If the function succeeds, the value returned is a pointer to the context of the operation. The returned context is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.10 CORBA::Request::contexts

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.17.10.1 Synopsis

Retrieves the context lists for the request.

## 14.17.10.2 C++ Binding

```
CORBA::ContextList ptr CORBA::Request::contexts ();
```

### 14.17.10.3 Arguments

None.

## 14.17.10.4 Description

This member function retrieves the context lists for the request.

### 14.17.10.5 Return Values

If the function succeeds, the value returned is a pointer to the context lists for the operation. The returned context list is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

## 14.17.11 CORBA::Request::exceptions

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

# 14.17.11.1 Synopsis

Retrieves the exception lists for the request.

# 14.17.11.2 C++ Binding

```
CORBA::ExceptionList ptr CORBA::Request::exceptions ();
```

## 14.17.11.3 Arguments

None.

# 14.17.11.4 Description

This member function retrieves the exception lists for the request.

### 14.17.11.5 Return Values

If the function succeeds, the value returned is a pointer to the exception list for the request. The returned exception list is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.12 CORBA::Request::target

Synopsis



- C++ Binding
- Arguments
- Description
- Return Values

### 14.17.12.1 Synopsis

Retrieves the target object reference for the request.

### 14.17.12.2 C++ Binding

```
CORBA::Object_ptr CORBA::Request::target () const;
```

# 14.17.12.3 Arguments

None.

### 14.17.12.4 Description

This member function retrieves the target object reference for the request.

### 14.17.12.5 Return Values

If the function succeeds, the value returned is a pointer to the target object of the operation. The returned value is owned by the Request object and must not be released.

If the function does not succeed, an exception is thrown.

# 14.17.13 CORBA::Request::send deferred

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

# 14.17.13.1 Synopsis

Initiates a deferred synchronous request.

# 14.17.13.2 C++ Binding

```
void CORBA::Request::send deferred ();
```

## 14.17.13.3 Arguments

None.

### 14.17.13.4 Description

This member function initiates a deferred synchronous request. You use this function when a response is expected and in conjunction with the CORBA::Request::get\_response function.

### 14.17.13.5 Return Values

None.

#### 14.17.13.6 See Also

```
CORBA::ORB::get_next_response

CORBA::ORB::poll_next_response

CORBA::ORB::send_multiple_requests

CORBA::Request::get_response

CORBA::Request::poll_response

CORBA::Request::send_oneway
```

# 14.17.14 CORBA::Request::send\_oneway

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values
- See Also

## 14.17.14.1 Synopsis

Initiates a one-way request.

# 14.17.14.2 C++ Binding

```
void CORBA::Request::send oneway ();
```

## 14.17.14.3 Arguments

None.

# 14.17.14.4 Description

This member function initiates a one-way request; it does not expect a response.

### 14.17.14.5 Return Values

None.

### 14.17.14.6 See Also

```
CORBA::ORB::send_multiple_requests
CORBA::Request::send_deferred
```

# 14.18 Strings

The mapping of these functions to C++ is as follows:

```
// C++
namespace CORBA {
  static char * string_alloc(ULong len);
  static char * string_dup (const char *);
  static void string_free(char *);
  ...
}
```

#### Note:

A static array of char in C++ decays to a char\*. Therefore, care must be taken when assigning a static array to a String\_var, because the String\_var assumes that the pointer points to data allocated via string\_alloc, and thus eventually attempts to free it using string free.

This behavior has changed in ANSI/ISO C++, where string literals are const char\*, not char\*. However, since most C++ compilers do not yet implement this change, portable programs must heed the advice given here.

The following sections describe the functions that manage memory allocated to strings.

- CORBA::string\_alloc
- CORBA::string\_dup
- CORBA::string\_free

## 14.18.1 CORBA::string alloc

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example
- See Also



### 14.18.1.1 Synopsis

Allocates memory for a string.

## 14.18.1.2 C++ Binding

```
char * CORBA::string alloc(ULong len);
```

### 14.18.1.3 Argument

len

The length of the string for which to allocate memory.

# 14.18.1.4 Description

This member function dynamically allocates memory for a string, or returns a nil pointer if it cannot perform the allocation. It allocateslen+1 characters so that the resulting string has enough space to hold a trailing NULL character. Free the memory allocated by this member function by calling the CORBA::string free member function.

This function does not throw CORBA exceptions.

### 14.18.1.5 Return Values

If the function succeeds, the return value is a pointer to the newly allocated memory for the string object; if the function fails, the return value is a nil pointer.

### 14.18.1.6 Example

```
char* s = CORBA::string alloc(10);
```

### 14.18.1.7 See Also

```
CORBA::string_free
CORBA::string dup
```

# 14.18.2 CORBA::string\_dup

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example
- See Also



# 14.18.2.1 Synopsis

Makes a copy of a string.

## 14.18.2.2 C++ Binding

```
char * CORBA::string dup (const char * Str);
```

## 14.18.2.3 Argument

Str

The address of the string to be copied.

## 14.18.2.4 Description

This function dynamically allocates enough memory to hold a copy of its string argument, including the NULL character, copies the string argument into that memory, and returns a pointer to the new string.

This function does not throw CORBA exceptions.

### 14.18.2.5 Return Values

If the function succeeds, the return value is a pointer to the new string; if the function fails, the return value is a nil pointer.

# 14.18.2.6 Example

```
char* s = CORBA::string_dup("hello world");
```

### 14.18.2.7 See Also

```
CORBA::string_free
CORBA::string alloc
```

# 14.18.3 CORBA::string\_free

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values
- Example
- See Also



### 14.18.3.1 Synopsis

Frees memory allocated to a string.

## 14.18.3.2 C++ Binding

```
void CORBA::string free(char * Str);
```

### 14.18.3.3 Argument

Str

The address of the memory to be deallocated.

## 14.18.3.4 Description

This member function deallocates memory that was previously allocated to a string using the  $CORBA::string\_alloc()$  or  $CORBA::string\_dup()$  member functions. Passing a nil pointer to this function is acceptable and results in no action being performed.

This function may not throw CORBA exceptions.

#### 14.18.3.5 Return Values

None.

### 14.18.3.6 Example

```
char* s = CORBA::string_dup("hello world");
CORBA::string free(s);
```

### 14.18.3.7 See Also

```
CORBA::string_alloc
CORBA::string dup
```

# 14.19 Wide Strings

Both bounded and unbounded wide string types are mapped to <code>CORBA::WChar\*</code> in C++. In addition, the CORBA module defines <code>WString\_var</code> and <code>WString\_out</code> classes. Each of these classes provides the same member functions with the same semantics as their string counterparts, except of course they deal with wide strings and wide characters.

Dynamic allocation and deallocation of wide strings must be performed via the following functions:

```
// C++
namespace CORBA {
// ...
WChar *wstring_alloc(ULong len);
WChar *wstring dup(const WChar* ws);
```



```
void wstring_free(WChar*);
};
```

These member functions have the same semantics as the same functions for the string type, except they operate on wide strings.

A compliant mapping implementation provides overloaded operator<< (insertion) and operator>> (extraction) operators for using WString\_var and WString\_out directly with C++ iostreams.

For descriptions of these member functions, see the corresponding function for Strings.

The following code snippet illustrates a code example that uses wide strings and wide characters.

```
// Get a string from the user:
cout << "String?";</pre>
char mixed[256]; // this should be big enough!
char lower[256];
char upper[256];
wchar t wmixed[256];
cin >> mixed;
// Convert the string to a wide char string,
// because this is what the server will expect.
mbstowcs (wmixed, mixed, 256);
// Convert the string to upper case:
CORBA::WString var v upper = CORBA::wstring dup(wmixed);
v simple->to upper(v upper.inout());
wcstombs(upper, v upper.in(), 256);
cout << upper << endl;</pre>
// Convert the string to lower case:
CORBA::WString var v lower = v simple->to lower(wmixed);
wcstombs(lower, v lower.in(), 256);
cout << lower << endl;</pre>
// Everything succeeded:
return 0;
```

# 14.20 TypeCode Member Functions

A TypeCode represents OMG IDL type information.

No constructors for TypeCodes are defined. However, in addition to the mapped interface, for each basic and defined OMG IDL type, an implementation provides access to a TypeCode pseudo-object reference (TypeCode\_ptr) of the form  $_tc_<type>$  that may be used to set types in Any, as arguments for equal, and so on. In the names of these TypeCode reference constants, <type> refers to the local name of the type within its defining scope. Each C++  $_tc_<type>$  constant is defined at the same scoping level as its matching type.

Like all other serverless objects, the C++ mapping for TypeCode provides a \_nil() operation that returns a nil object reference for a TypeCode. This operation can be used to initialize TypeCode references embedded within constructed types. However, a nil TypeCode reference may never be passed as an argument to an operation, since TypeCodes are effectively passed as values, not as object references.

The mapping of these member functions to C++ is as follows:

```
class CORBA
{
  class TypeCode
  {
  public:
    class Bounds { ... };
    class BadKind { ... };

Boolean equal(TypeCode_ptr) const;
TCKind kind() const;
Long param_count() const;
Any *parameter(Long) const;
RepositoryId id () const;
}; // TypeCode
}; // CORBA
```

Memory Management

# 14.20.1 Memory Management

TypeCode has the following special memory management rule:

 Ownership of the return values of the id function is maintained by the TypeCode; these return values must not be freed by the caller.

The following sections describe these member functions.

- CORBA::TypeCode::equal
- CORBA::TypeCode::id
- CORBA::TypeCode::kind
- CORBA::TypeCode::param count
- CORBA::TypeCode::parameter

# 14.20.1.1 CORBA::TypeCode::equal

- Synopsis
- C++ Binding
- Argument
- Description
- Return Values

### 14.20.1.1.1 Synopsis

Determines whether two TypeCode objects are equal.

## 14.20.1.1.2 C++ Binding

```
CORBA::Boolean CORBA::TypeCode::equal (
CORBA::TypeCode_ptr TypeCodeObj) const;
```

### 14.20.1.1.3 Argument

#### TypeCodeObj

A pointer to a TypeCode object with which to make the comparison.

### 14.20.1.1.4 Description

This member function determines whether a TypeCode object is equal to the input parameter, TypeCodeObj.

#### 14.20.1.1.5 Return Values

If the TypeCode object is equal to the TypeCodeObj parameter, CORBA TRUE is returned.

If the TypeCode object is not equal to the TypeCodeObj parameter, CORBA FALSE is returned.

If the function does not succeed, an exception is thrown.

### 14.20.1.2 CORBA::TypeCode::id

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.20.1.2.1 Synopsis

Returns the ID for the TypeCode.

## 14.20.1.2.2 C++ Binding

CORBA::RepositoryId CORBA::TypeCode::id () const;

### 14.20.1.2.3 Arguments

None.

# 14.20.1.2.4 Description

This member function returns the ID for the TypeCode.

### 14.20.1.2.5 Return Values

Repository ID for the TypeCode.

# 14.20.1.3 CORBA::TypeCode::kind

- Synopsis
- C++ Binding
- Arguments



- Description
- Return Values

### 14.20.1.3.1 Synopsis

Retrieves the kind of data contained in the TypeCode object reference.

# 14.20.1.3.2 C++ Binding

```
CORBA::TCKind CORBA::TypeCode::kind () const;
```

#### 14.20.1.3.3 Arguments

None.

### 14.20.1.3.4 Description

This member function retrieves the kind attribute of the CORBA:: TypeCode class, which specifies the kind of data contained in the TypeCode object reference.

#### 14.20.1.3.5 Return Values

If the member function succeeds, it returns the kind of data contained in the TypeCode object reference. For a list of the TypeCode kinds and their parameters, see the following table.

If the member function does not succeed, an exception is thrown.

Table 14-3 Legal Typecode Kinds and Parameters

| TypeCode Kind       | Parameters List   |
|---------------------|---|
| CORBA::tk_null      | *NONE*  |
| CORBA::tk_void      | *NONE*  |
| CORBA::tk_short     | *NONE*  |
| CORBA::tk_long      | *NONE*  |
| CORBA::tk_long      | *NONE*  |
| CORBA::tk_ushort    | *NONE*  |
| CORBA::tk_ulong     | *NONE*  |
| CORBA::tk_float     | *NONE*  |
| CORBA::tk_double    | *NONE*  |
| CORBA::tk_boolean   | *NONE*  |
| CORBA::tk_char      | *NONE*  |
| CORBA::tk_wchar     | *NONE*  |
| CORBA::tk_octet     | *NONE*  |
| CORBA::tk_Typecode  | *NONE*  |
| CORBA::tk_Principal | *NONE*  |
| CORBA::tk_objref    | {interface_id}  |
| CORBA::tk_struct    | <pre>{struct-name, member-name, TypeCode, (repeat pairs)}</pre>               |
| CORBA::tk_union     | <pre>{union-name, switch-TypeCode, label- value, member-name, enum-id,}</pre> |

Table 14-3 (Cont.) Legal Typecode Kinds and Parameters

| TypeCode Kind      | Parameters List            |
|--------------------|----------------------------|
| CORBA::tk_enum     | {enum-name, enum-id,}      |
| CORBA::tk_string   | {maxlen-integer}           |
| CORBA::tk_wstring  | {maxlen-integer}           |
| CORBA::tk_sequence | {TypeCode, maxlen-integer} |
| CORBA::tk_array    | {TypeCode, length-integer} |

## 14.20.1.4 CORBA::TypeCode::param count

- Synopsis
- C++ Binding
- Arguments
- Description
- Return Values

### 14.20.1.4.1 Synopsis

Retrieves the number of parameters for the TypeCode object reference.

### 14.20.1.4.2 C++ Binding

```
CORBA::Long CORBA::TypeCode::param count () const;
```

### 14.20.1.4.3 Arguments

None.

### 14.20.1.4.4 Description

This member function retrieves the parameter attribute of the CORBA::TypeCode class, which specifies the number of parameters for the TypeCode object reference. For a list of parameters of each kind, see the table *Legal Typecode Kinds and Parameters*.

#### 14.20.1.4.5 Return Values

If the function succeeds, it returns the number of parameters contained in the TypeCode object reference.

If the function does not succeed, an exception is thrown.

# 14.20.1.5 CORBA::TypeCode::parameter

- Synopsis
- C++ Binding
- Argument
- Description

#### Return Values

#### 14.20.1.5.1 Synopsis

Retrieves a parameter specified by the index input argument.

#### 14.20.1.5.2 C++ Binding

```
CORBA::Any * CORBA::TypeCode::parameter (
CORBA::Long Index) const;
```

### 14.20.1.5.3 Argument

#### Index

An index to the parameter list, used to determine which parameter to retrieve.

#### 14.20.1.5.4 Description

This member function retrieves a parameter specified by the index input argument. For a list of parameters of each kind, see the table *Legal Typecode Kinds and Parameters*.

#### 14.20.1.5.5 Return Values

If the member function succeeds, the return value is a pointer to the parameter specified by the index input argument.

If the member function does not succeed, an exception is thrown.

# 14.21 Exception Member Functions

The Oracle Tuxedo software supports the throwing and catching of exceptions.

Caution: Use of the wrong exception constructor causes non initialization of a data member. Exceptions that are defined to have a reason field need to be created using the constructor that initializes that data member. If the default constructor is used instead, that data member is not initialized and, during destruction of the exception, the system may attempt to destroy nonexistent data. When creating exceptions, be sure to use the constructor function that most fully initializes the data fields. These exceptions can be most easily identified by looking at the OMG IDL definition; they have additional data member definitions.

Descriptions of exception member functions follow:

```
CORBA::SystemException::SystemException ()
```

This is the default constructor for the CORBA::SystemException class. Minor code is initialized to 0 (zero) and the completion status is set to COMPLETED NO.

CORBA::SystemException::SystemException ( const CORBA::SystemException & Se) This is the copy constructor for the CORBA::SystemException class.

```
CORBA::SystemException::SystemException( CORBA::ULong Minor,
CORBA::CompletionStatus Status)
```

This constructor for the CORBA::SystemException class sets the minor code and completion status.

Explanations of the arguments are as follows:

#### Minor

The minor code for the Exception object. The minor field is an implementation-specific value used by the ORB to identify the exception. The Oracle Tuxedo minor field definitions can be found in the file orbminor.h.

#### Status

The completion status for the Exception object. The values are as follows:

CORBA::COMPLETED\_YES
CORBA::COMPLETED\_NO
CORBA::COMPLETED MAYBE

CORBA::SystemException::~SystemException ()

This is the destructor for the CORBA::SystemException class. It frees any memory used for the Exception object.

CORBA::SystemException CORBA::SystemException::operator = const CORBA::SystemException Se)

This assignment operator copies exception information from the source exception. The Se argument specifies the SystemException object that is to be copied by this operator.

CORBA::CompletionStatus CORBA::SystemException::completed()
This member function returns the completion status for this exception.

CORBA::SystemException::completed( CORBA::CompletionStatus Completed)

This member function sets the completion status for this exception. The Completed argument specifies the completion status for this exception.

CORBA::ULong CORBA::SystemException::minor()

This member function returns the minor code for this exception.

CORBA::SystemException::minor (CORBA::ULong Minor)

This member function sets the minor code for this exception. The minor argument specifies the new minor code for this exception. The minor field is an implementation-specific value used by the application to identify the exception.

CORBA::SystemException \* CORBA::SystemException::\_narrow ( CORBA::Exception\_ptr Exc)

This member function determines whether a specified exception can be narrowed to a system exception. The Exc argument specifies the exception to be narrowed.

If the specified exception is a system exception, this member function returns a pointer to the system exception. If the specified exception is not a system exception, the function returns 0 (zero).

CORBA::UserException \* CORBA::UserException::\_narrow( CORBA::Exception\_ptr Exc) This member function determines whether a specified exception can be narrowed to a user exception. The Exc argument specifies the exception to be narrowed.

If the specified exception is a user exception, this member function returns a pointer to the user exception. If the specified exception is not a user exception, the function returns 0 (zero).

# 14.22 Standard Exceptions

This section presents the standard exceptions defined for the ORB. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions are not listed in raises expressions.

To bound the complexity in handling the standard exceptions, the set of standard exceptions is kept to a tractable size. This constraint forces the definition of equivalence classes of exceptions, rather than enumerating many similar exceptions.

For example, an operation invocation can fail at many different points due to the inability to allocate dynamic memory. Rather than enumerate several different exceptions that correspond to the different ways that memory allocation failure causes the exception (during marshaling, unmarshaling, in the client, in the object implementation, allocating network packets, and so forth), a single exception corresponding to dynamic memory allocation failure is defined. Each standard exception includes a minor code to designate the subcategory of the exception; the assignment of values to the minor codes is left to each ORB implementation.

Each standard exception also includes a <code>completion\_status</code> code, which takes one of the following values:

CORBA::COMPLETED YES

The object implementation completed processing prior to the exception being raised.

CORBA::COMPLETED NO

The object implementation was not initiated prior to the exception being raised.

CORBA::COMPLETED MAYBE

The status of implementation completion is unknown.

- Exception Definitions
- Object Nonexistence
- Transaction Exceptions

# 14.22.1 Exception Definitions

The standard exceptions are defined below. Clients must be prepared to handle system exceptions that are not on this list, both because future versions of this specification may define additional standard exceptions, and because ORB implementations may raise nonstandard system exceptions. For more information about exceptions, see *System Messages*.

The following table defines the exceptions.

**Table 14-4 Exception Definitions** 

| Exception           | Description                        |
|---------------------|------------------------------------|
| CORBA::UNKNOWN      | The unknown exception.             |
| CORBA::BAD_PARAM    | An invalid parameter was passed.   |
| CORBA::NO_MEMORY    | Dynamic memory allocation failure. |
| CORBA::IMP_LIMIT    | Violated implementation limit.     |
| CORBA::COMM_FAILURE | Communication failure.             |



Table 14-4 (Cont.) Exception Definitions

| Exception                     | Description                            |  |
|-------------------------------|--|--|
| CORBA::INV_OBJREF             | Invalid object reference.              |  |
| CORBA::NO_PERMISSION          | No permission for attempted operation. |  |
| CORBA::INTERNAL               | ORB internal error.                    |  |
| CORBA::MARSHAL                | Error marshalling parameter/result.    |  |
| CORBA::INITIALIZE             | ORB initialization failure.            |  |
| CORBA::NO_IMPLEMENT           | Operation implementation unavailable.  |  |
| CORBA::BAD_TYPECODE           | Bad typecode.                          |  |
| CORBA::BAD_OPERATION          | Invalid operation.                     |  |
| CORBA::NO_RESOURCES           | Insufficient resources for request.    |  |
| CORBA::NO_RESPONSE            | Response to request not yet available. |  |
| CORBA::PERSIST_STORE          | Persistent storage failure.            |  |
| CORBA::BAD_INV_ORDER          | Routine invocations out of order.      |  |
| CORBA::TRANSIENT              | Transient failure; reissue request.    |  |
| CORBA::FREE_MEM               | Cannot free memory.                    |  |
| CORBA::INV_IDENT              | Invalid identifier syntax.             |  |
| CORBA::INV_FLAG               | Invalid flag was specified.            |  |
| CORBA::INTF_REPOS             | Error accessing interface repository.  |  |
| CORBA::BAD_CONTEXT            | Error processing context object.       |  |
| CORBA::OBJ_ADAPTER            | Failure detected by object adapter.    |  |
| CORBA::DATA_CONVERSION        | Data conversion error.                 |  |
| CORBA::OBJECT_NOT_EXIST       | Nonexistent object; delete reference.  |  |
| CORBA::TRANSACTION_REQUIRED   | Transaction required.                  |  |
| CORBA::TRANSACTION_ROLLEDBACK | Transaction rolled back.               |  |
| CORBA::INVALID_TRANSACTION    | Invalid transaction.                   |  |

# 14.22.2 Object Nonexistence

The CORBA::OBJECT\_NOT\_EXIST exception is raised whenever an invocation on a deleted object is performed. It is an authoritative "hard" fault report. Anyone receiving it is allowed (even expected) to delete all copies of this object reference and to perform other appropriate "final recovery" style procedures.

# 14.22.3 Transaction Exceptions

The <code>CORBA::TRANSACTION\_REQUIRED</code> exception indicates that the request carried a <code>NULL</code> transaction context, but an active transaction is required.

The <code>CORBA::TRANSACTION\_ROLLEDBACK</code> exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

The CORBA::INVALID\_TRANSACTION indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occurred when trying to register a resource.

# 14.23 ExceptionList Member Functions

The ExceptionList member functions allow a client or server application to provide a list of TypeCodes for all user-defined exceptions that may result when the Request is invoked. For a description of the Request member functions, see the section Request Member Functions.

The mapping of these member functions to C++ is as follows:

```
class CORBA
{
    class ExceptionList
    {
        public:
            Ulong count();
        void add(TypeCode_ptr tc);
        void add_consume(TypeCode_ptr tc);
        TypeCode_ptr item(Ulong index);
        Status remove(Ulong index);
    }; // ExceptionList
} // CORBA
```

- CORBA::ExceptionList::count
- CORBA::ExceptionList::add
- CORBA::ExceptionList::add\_consume
- CORBA::ExceptionList::item
- <Enter Topic Title Here>
   (Required) <Enter a short description here.>

# 14.23.1 CORBA::ExceptionList::count

- Synopsis
- C++ Binding
- Arguments
- Exception
- Description
- Return Values

# 14.23.1.1 Synopsis

Retrieves the current number of items in the list.

## 14.23.1.2 C++ Binding

```
Ulong count ();
```

### 14.23.1.3 Arguments

None.

## 14.23.1.4 Exception

If the function does not succeed, an exception is thrown.

## 14.23.1.5 Description

This member function retrieves the current number of items in the list.

#### 14.23.1.6 Return Values

If the function succeeds, the returned value is the number of items in the list. If the list has just been created, and no ExceptionList objects have been added, this function returns 0 (zero).

# 14.23.2 CORBA::ExceptionList::add

- Synopsis
- C++ Binding
- Arguments
- Exception
- Description
- Return Values
- See Also

# 14.23.2.1 Synopsis

Constructs a ExceptionList object with an unnamed item, setting only the flags attribute.

# 14.23.2.2 C++ Binding

void add(TypeCode ptr tc);

## 14.23.2.3 Arguments

tc

Defines the memory location referred to by TypeCode ptr.

### 14.23.2.4 Exception

If the member function does not succeed, a CORBA::NO MEMORY exception is thrown.

# 14.23.2.5 Description

This member function constructs an ExceptionList object with an unnamed item, setting only the flags attribute.

The ExceptionList object grows dynamically; your application does not need to track its size.

#### 14.23.2.6 Return Values

If the function succeeds, the return value is a pointer to the newly created ExceptionList object.

#### 14.23.2.7 See Also

```
CORBA::ExceptionList::add_consume
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove
```

# 14.23.3 CORBA::ExceptionList::add\_consume

- Synopsis
- C++ Binding
- Arguments
- Exceptions
- Description
- Return Values
- See Also

# 14.23.3.1 Synopsis

Constructs an ExceptionList object.

## 14.23.3.2 C++ Binding

```
void add consume(TypeCode ptr tc);
```

# 14.23.3.3 Arguments

tc

The memory location to be assumed.

# 14.23.3.4 Exceptions

If the member function does not succeed, an exception is raised.

## 14.23.3.5 Description

This member function constructs an ExceptionList object.

The ExceptionList object grows dynamically; your application does not need to track its size.

### 14.23.3.6 Return Values

If the function succeeds, the return value is a pointer to the newly created ExceptionList object.

### 14.23.3.7 See Also

CORBA::ExceptionList::add
CORBA::ExceptionList::count
CORBA::ExceptionList::item
CORBA::ExceptionList::remove

# 14.23.4 CORBA::ExceptionList::item

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

# 14.23.4.1 Synopsis

Retrieves a pointer to the ExceptionList object, based on the index passed in.

## 14.23.4.2 C++ Binding

TypeCode ptr item(ULong index);

## 14.23.4.3 Argument

#### index

The index into the ExceptionList object. The indexing is zero-based.

### 14.23.4.4 Exceptions

If the function does not succeed, the BAD PARAM exception is thrown.

## 14.23.4.5 Description

This member function retrieves a pointer to an ExceptionList object, based on the index passed in. The function uses zero-based indexing.

## 14.23.4.6 Return Values

If the function succeeds, the return value is a pointer to the ExceptionList object.

### 14.23.4.7 See Also

CORBA::ExceptionList::add

CORBA::ExceptionList::add consume

CORBA::ExceptionList::count CORBA::ExceptionList::remove CORBA::ExceptionList::remove

# 14.23.5 < Enter Topic Title Here>

(Required) <Enter a short description here.>

<Enter a single subject here.>

- Synopsis
- C++ Binding
- Argument
- Exceptions
- Description
- Return Values
- See Also

# 14.23.5.1 Synopsis

Removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

### 14.23.5.2 C++ Binding

Status remove(ULong index);

# 14.23.5.3 Argument

#### Index

The index into the ContextList object. The indexing is zero-based.

# 14.23.5.4 Exceptions

If the function does not succeed, the  ${\tt BAD}\,$  param exception is thrown.

# 14.23.5.5 Description

This member function removes the item at the specified index, frees any associated memory, and reorders the remaining items on the list.

### 14.23.5.6 Return Values

None.

### 14.23.5.7 See Also

CORBA::ExceptionList::add

CORBA::ExceptionList::add\_consume

CORBA::ExceptionList::count
CORBA::ExceptionList::item



15

# Server Side Mapping

Server-side mapping refers to the portability constraints for an object implementation written in C++. The term *server* is not meant to restrict implementations to situations in which method invocations cross-address space or machine boundaries. This mapping addresses any implementation of an Object Management Group (OMG) Interface Definition Language (IDL) interface.

#### Note:

The information in this chapter is based on the *Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, February 2001, published by the Object Management Group (OMG). Used with permission of the OMG.

- Implementing Interfaces
- Inheritance-based Interface Implementation
- Delegation-based Interface Implementation
- Implementing Operations

# 15.1 Implementing Interfaces

To define an implementation in C++, you define a C++ class with any valid C++ name. For each operation in the interface, the class defines a non-static member function with the mapped name of the operation (the mapped name is the same as the OMG IDL identifier).

The server application mapping specifies two alternative relationships between the implementation class supplied by the application and the generated class or classes for the interface. Specifically, the mapping requires support for both inheritance-based relationships and delegation-based relationships. Conforming applications may use either or both of these alternatives. Oracle Tuxedo CORBA supports both inheritance-based and delegation-based relationships.

# 15.2 Inheritance-based Interface Implementation

In the inheritance-based interface implementation approach, the implementation classes are derived from a generated base class based on the OMG IDL interface definition. The generated base classes are known as skeleton classes, and the derived classes are known as implementation classes. Each operation of the interface has a corresponding virtual member function declared in the skeleton class. The generated skeleton class is partially opaque to the programmer, though it will contain a member function corresponding to each operation in the interface. The signature of the member function is identical to that of the generated client stub class.

To implement this interface using inheritance, a programmer must derive from this skeleton class and implement each of the operations in the OMG IDL interface. To allow portable implementations to multiple inheritances from both skeleton classes and implementation

classes for other base interfaces without error or ambiguity, the <code>Tobj\_ServantBase</code> class must be a virtual base class of the skeleton, and the <code>PortableServer::ServantBase</code> class must be a virtual base class of the <code>Tobj\_ServantBase</code> class. The inheritance among the implementation class, the <code>Skeleton</code> class, the <code>Tobj\_ServantBase</code> class, and the <code>PortableServer::ServantBase</code> class must all be public virtual.

The implementation class or servant must only derive directly from a single generated skeleton class. Direct derivation from multiple skeleton classes could result in ambiguous errors due to multiple definitions of the <code>\_this()</code> operation. This should not be a limitation, however, since CORBA objects have only a single most-derived interface. C++ servants that are intended to support multiple interface types can utilize the delegation-based interface implementation approach. See the following code snippet for an example of OMG IDL that uses interface inheritance.

```
// IDL
interface A
{
    short op1();
    void op2(in long val);
};
```

The following code snippet for an example of Interface Class A.

```
// C++
class A : public virtual CORBA::Object
{
    public:
        virtual CORBA::Short op1 ();
        virtual void op2 (CORBA::Long val);
};
```

On the server side, a skeleton class is generated. This class is partially opaque to the programmer, though it does contain a member function corresponding to each operation in the interface.

For the Portable Object Adapter (POA), the name of the skeleton class is formed by prepending the string "POA\_" to the fully scoped name of the corresponding interface, and the class is directly derived from the servant base class Tobj\_ServantBase. The C++ mapping for Tobj ServantBase is as follows:

```
// C++
class Tobj_ServantBase
{
   public:
      virtual void activate_object(const char* stroid);
      virtual void deactivate_object (
            const char* stroid,
            TobjS::DeactivateReasonValue reason
     );
}
```

The activate\_object() and deactivate\_object() member functions are described in detail in the sections Tobj\_ServantBase:: activate\_object() and Tobj\_ServantBase::\_add\_ref().

The skeleton class for interface A shown above would appear as shown in the following code snippet.

```
// C++
class POA_A : public Tobj_ServantBase
{
public:
    // ... server-side ORB-implementation-specific
    // goes here...
    virtual CORBA::Short op1 () = 0;
    virtual void op2 (CORBA::Long val) = 0;
    //...
};
```

If interface  $\mathbb{A}$  were defined within a module rather than at global scope (for example,  $\mathbb{Mod}: \mathbb{A}$ ), the name of its skeleton class would be  $\mathbb{POA}_{Mod}: \mathbb{A}$ . This helps to separate server application skeleton declarations and definitions from C++ code generated for the client.

To implement this interface using inheritance, you must derive from this skeleton class and implement each of the operations in the corresponding OMG IDL interface. An implementation class declaration for interface A would take the form shown in the following code snippet.

# 15.3 Delegation-based Interface Implementation

The delegation-based interface implementation approach is an alternative to using inheritance when implementing CORBA objects. This approach is used when the overhead of inheritance is too high or cannot be used. For example, due to the invasive nature of inheritance, implementing objects using existing legacy code might be impossible if inheritance for some global class were required. Instead, delegation can be used to solve these types of problems. Delegation is a more natural fit doing object implementations when the Process-Entity design pattern is used. In this pattern, the Process object would delegate operations onto one or more entity objects.

In the delegation-based approach, the implementation does not inherit from a skeleton class. Instead, the implementation can be coded as required for the application, and a wrapper object will delegate upcalls to that implementation. This "wrapper object," called a *tie*, is generated by the IDL compiler, along with the same skeleton class used for the inheritance approach. The generated *tie* class is partially opaque to the programmer, though, like the skeleton, it provides a method corresponding to each OMG IDL operation for the associated interface. The name of the generated *tie* class is the same as the generated skeleton class with the addition that the string tie is appended to the end of the class name.

An instance of the tie class is the servant, not the C++ object being delegated to by the tie object, that is passed as the argument to the operations that require a Servant argument. It

should also be noted that the tied object has no access to the  $\_this()$  operation, nor should it access data members directly.

A type-safe tie class is implemented using C++ templates. The following code snippet explains a tie class generated from the Derived interface in the previous OMG IDL example.

```
// C++
template <class T>
class POA A tie : public POA A {
public:
  POA A_tie(T& t)
      : ptr(&t), poa(PortableServer::POA:: nil()), rel(0) {}
  POA A tie (T& t, PortableServer:: POA ptr poa)
      : _ptr(&t), _poa(PortableServer::POA::_duplicate(poa)), rel(0) {}
   POA A tie(T* tp, CORBA::Boolean release = 1)
      : ptr(tp), poa(PortableServer::POA:: nil()), rel(release) {}
  POA A tie(T* tp, PortableServer::POA ptr poa, CORBA::Boolean release = 1)
      : ptr(tp), poa(PortableServer::POA:: duplicate(poa)), rel(release)
{ }
  ~POA A tie()
   { CORBA::release( poa);
  if ( rel) delete _ptr;
  // tie-specific functions
  T* tied object () {return ptr;}
  void tied object(T& obj)
{ if ( rel) delete ptr;
_ptr = &obj;
rel = 0;
void tied object(T* obj, CORBA::Boolean release = 1)
{ if ( rel) delete ptr;
ptr = obj;
rel = release;
CORBA::Boolean is owner() { return rel; }
void is owner (CORBA::Boolean b) { rel = b; }
// IDL operations*************
CORBA::Short op1 ()
  return ptr->op1 ();
void op2 (CORBA::Long val)
   ptr->op2 (val);
// *************
// override ServantBase operations
PortableServer::POA ptr default POA()
    if (!CORBA::is nil( poa))
        return poa;
    else {
#ifdef WIN32
```

This class definition is a template generated by the IDL compiler. You typically use it by first getting a pointer to the legacy class and then instantiating the tie class with that pointer. For example:

As you can see, the tie class contains definitions for the op1 and op2 operations of the interface that assume that the legacy class has operations with the same signatures as those given in the IDL. If this is the case, you can use the tie class file as is, letting it delegate exactly. It is more likely, however, that the legacy class will not have identical signatures or you may have to do more than a single function call. In that case, it is your job to replace the code for op1 and op2 in this generated code. The code for each operation typically makes invocations on the legacy class using the tie class variable <code>\_ptr</code>, which contains the pointer to the legacy class. For example, you might change the following lines:

```
CORBA::Short op1 () {return _ptr->op1 (); }
void op2 (CORBA::Long val) {_ptr->op2 (val); }

to the following:

CORBA::Short op1 ()
{
    return _ptr->op37 ();
}
void op2 (CORBA::Long val)
{
    CORBA::Long temp;
    temp = val + 15;
    _ptr->lookup(val, temp, 43);
}
```

An instance of this template class performs the task of delegation. When the template is instantiated with a class type that provides the operation of the <code>Derived</code> interface, then the <code>POA\_Derived\_tie</code> class will delegate all operations to an instance of that implementation class. A reference or pointer to the actual implementation object is passed to the appropriate tie constructor when an instance of the <code>POA\_Derived\_tie</code> class is created. When a request is

invoked on it, the tie servant will just delegate the request by calling the corresponding method on the implementation class.

The use of templates for tie classes allows the application developer to provide specializations for some or all of the template's operations for a given instantiation of the template. This allows the application to use legacy classes for tied object types, where the operation signatures of the tied object will differ from that of the tie class.

# 15.4 Implementing Operations

The signature of an implementation member function is the mapped signature of the OMG IDL operation. Unlike the client-side mapping, the OMG specifies that the function header for the server-side mapping include the appropriate exception specification. The following code snippet illustrates an example of an exception specification.

```
// IDL
interface A
{
    exception B {};
    void f() raises(B);
};
// C++
class MyA : public virtual POA_A
{
    public:
       void f();
       ...
};
```

Since all operations and attributes may raise CORBA system exceptions, CORBA::SystemException must appear in all exception specifications, even when an operation has no raises clause.

#### Note:

Because of the differences in C++ compilers, it is best to leave out the "throw declaration" in the method signature. Some systems cause the application server to crash if an undeclared exception is thrown in a method that has declared the exceptions it will throw.

Within a member function, the "this" pointer refers to the implementation object's data as defined by the class. In addition to accessing the data, a member function may implicitly call another member function defined by the same class. The following code snippet illustrates an example of calling another member function.

```
// IDL
interface A
{
    void f();
    void g();
};
// C++
class MyA : public virtual POA A
```

```
{
    public:
        void f();
        void g();
    private:
        long x_;
};
void
MyA::f();
{
        x_ = 3;
        g();
}
```

When a servant member function is invoked in this manner, it is being called simply as a C++ member function, not as the implementation of an operation on a CORBA object.